PackerGrind: An Adaptive Unpacking System for Android Apps

Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, Xiaobo Ma

Abstract—App developers are increasingly using packing services (or packers) to protect their code against being reverse engineered or modified. However, such packing techniques are also leveraged by the malicious developers to prevent the malware from being analyzed and detected by the static malware analysis and detection systems. Though there are already studies on unpacking packed Android apps, they usually leverage the manual reverse engineered packing behaviors to unpack apps packed by the specific packers and cannot be appified to the evolved and new packers. In this paper, we propose a novel unpacking approach with the capacity of adaptively unpacking the evolved and newly encountered packers. Also, we develop a new system, named `PackerGrind`, based on this adaptive approach for unpacking Android packers. The evaluation with real packed apps demonstrates that `PackerGrind` can successfully reveal packers' protection mechanisms, effectively handle their evolution and recover Dex files with low overhead.

1 Introduction

Android has become the most popular system and there are already 3.6 million apps available in the largest app store Google Play in March 2018 [17]. Meanwhile, Android apps have been the largest malware target [2], [42] and many malicious apps were constructed by injecting malicious code into normal Android apps and then repackaging them into new apps [28], [33], [58], [80]. Specifically, the adversaries usually leverage repacking techniques to embed malicious code into the legitimate apps through repacking. Note that, since the apps are not well protected,, such as packing or obfuscation, the adversaries can straightforwardly reverse engineer such apps and modify, which is already one of the OWASP mobile top ten risks [3].

Consequently, various packing services are provided for app authors with the purpose of preventing legitimate apps from being reverse engineered and repackaged [34]. However, not only the legitimate app authors leverage these packing services to protect their own apps but the adversaries also employ such services to hide the contents of the malicious apps and prevent the malicious apps from being analyzed and detected statically [21], [44], [75], [78], [81]. Even worse, with the increasing popularity of packing services, the percentage of packed malicious apps has also increased from 10% to 25% according to the report of Symantec [18]. To address this issue, multiple tools have been proposed to expose the hidden payload in the packed Android apps [31], [50], [75], [78].

With the evolving of the packing techniques, such tools cannot be applied to unpacking the apps packed by the latest and evolved packers, because these tools do not adaptively unpack the packed apps through dynamically tracking packing behaviors. Hence, with the purpose of unpacking Android apps through adaptively monitoring the packing behaviors, we propose a novel approach to adaptively unpacking packed apps. Also, we design and implement the unpacking system `PackerGrind` to facilitate such unpacking procedure.

Given a new encountered packed Android app, our adaptive unpacking approach unpacks it through three major phases, including dynamic monitoring, Dex recovery, and static analysis. First, we leverage *dynamic monitoring* to track the packing behaviors of the packed apps, especially the information about the release of the hidden content, the execution of the content in memory, and the location of the released content. All the monitored information will be stored in the log file, and then we can determine Dex data collection points through analyzing the log files. Second, we dynamically dump the Dex data in the memory at the determined Dex data collection points and then recover the dumped Dex data into a valid Dex file finally. Third, we also determine whether the recovered Dex file contains all required Dex data through statically analyzing the Dex file and the log files. If more Dex data are required, we repeat the above unpacking phases to determine the data collection points for the new data and finally recover a new Dex file based on the collected Dex data. Note that, for each packer, we store the required Dex collection points in a configuration file and thus we can directly unpack and recover the Dex file at the corresponding points if the packer is already analyzed before.

Because of various challenges, it is nontrivial to leverage the adaptive unpack approach to unpack the packed apps automatically. **C1:** It is challenging to monitor the cross-layer packing behaviors of Android packers on the real smartphone. **C2:** It is challenging to unpack and recover the protected Dex files with high efficiency. **C3:** Recognizing the packers that are already analyzed is also challenging.

We need to perform behavior tracking in multiple layers because the Android system consists of multiple layers and adopts two types of runtime, DVM (Dalvik Virtual Machine) and ART Android runtime, which are used before

- *Lei Xue, Hao Zhou, Xiapu Luo, and Le Yu are with Department of Computing at the Hong Kong Polytechnic University;*
- *Dinghao Wu is with College of Information Sciences and Technology, The Pennsylvania State University;*
- *Yajin Zhou is with College of Computer Science and Technology, Zhejiang University;*
- *Xiaobo Ma is with Department of Computer Science and Technology, Xi'an Jiaotong University*

and from Android 5.0 by default, respectively. In addition, since the existing analysis and unpacking tools usually conducts dynamic analysis based on Android emulator (i.e., Qemu) [31], [63], [73] or debugging techniques (e.g., *ptrace()*) [74], [79], the sophisticated Android packers can detect emulator and debugging environment to prevent them from being detected and analyzed. Hence, to address challenge **C1**, we propose a novel cross-layer (i.e., framework layer, runtime layer, and system layer) tracking mechanism to monitor the packing behaviors of Android packers on the real smartphone by leveraging the DBI (dynamic binary instrumentation) framework Valgrind [48].

In addition, since the existing unpacking tools leverage the one-process strategy to unpack packed Android apps, they cannot fully address challenge **C2**. To address this issue, we propose an approach to collect the Dex data at the points determined according to the packing behaviors because different packers adopt various protection mechanisms to protect the Dex data whereas they all need to dynamically release the protected data into memory. `PackerGrind` dynamically monitors the details about the packing behaviors, especially those related to the Dex data releasing and modification, and then determines the Dex data collection points through carefully analyzing the packing behaviors stored in the log files according to our suggested criteria. Finally, `PackerGrind` disassembles the collected Dex data into a new Dex file.

Given a packed app, if its packer can be identified and the samples protected by the same packer have been analyzed before, the unpacking process for the previous samples can be applied to the new packed app for speeding up the unpacking. Although DexHunter also tries to identify packers by searching specific files in the file system or special strings in memory, the packers can easily change the file names and strings to impede the unpacking. To address challenge **C3**, `PackerGrind` recognizes packers according to their initialization behaviors that will not be changed frequently.

In summary, the main contributions of this paper include:

- We propose a novel approach to adaptively unpack the Android apps through tracking the packers' dynamic packing behaviors and determining the Dex data collection points, and thus this approach can also be applied to the new encountered or evolved Android packers.
- To facilitate such adaptive unpacking procedure, we also design and implement the unpacking system `PackerGrind`[1], which runs in the real smartphone and dynamically tracks the packing behaviors in multiple layers. Also, it can recognize the analyzed packers and thus it just needs to track the packing behaviors once for each Android packer.
- We evaluate `PackerGrind` leveraging apps packed by various commercial packers and compare it with the existing unpackers. The evaluation results show that `PackerGrind` successfully unpacks all packed apps and outperforms the other popular unpackers.

This paper is an extension of [71] and the rest of it is organized as follows. We first introduce the background as well as a motivating example in Section 2, and then describe the basic Dex data collection points in Section 4. The design and implementation of `PackerGrind` are detailed in Section 5 and the evaluation results are presented in Section 6. Afterwards, Section 7 presents our discussion about the limitations of `PackerGrind` as well as the future work. Then, after presenting the related work in Section 8, we conclude our work in Section 9.

1. https://github.com/rewhy/adaptiveunpacker

2 Background

In this section, we will introduce the necessary background information about the Dex file, Oat file, Android runtime, and the packing techniques.

2.1 Dex File

All data in Dex file are organized following specific Dex file format and the Dex file consists of two separate major portions, the Dex header portion storing the metadata and the body portion containing the majority of the data. Particularly, the metadata stored in the Dex header includes the Dex file magic, file checksum, SHA1 signature, file size, header size, endian constant, as well as the sizes and locations of the data structures that hold the identifiers for the methods, strings, and other items in the Dex file. Whereas, the data portion contains the data related to the implementations of the classes, methods, and the bytecode, such as the identifiers pointing to various structures and the structures storing the bytecode of each method. Consequently, the static analysis is always carried out through disassembling the Dex files contained in the app. Hence, the purpose of this paper is to expose the hidden implementations of the packed apps and then reassemble them into a valid Dex file so as to facilitate other static analysis tools.

2.2 Oat File

For the ART runtime, different from the Dex file contained in the Android app, the Oat file is generated when the bytecode in the Dex file is compiled into native code by the runtime tool `dex2oat` and all the generated native code as well as the original bytecode is stored in the Oat file. The Oat file is actually an extended ELF file [65] and consists of multiple data sections, which stores various types of data, such as the original Dex files, the native code, and the metadata. Note that, for each compiled method, both its native code and bytecode are stored in different data structures in the Oat file but the ART runtime also has the capacity of interpreting the methods that contain no native code.

2.3 Android Runtime

For existing Android systems, there are two types of runtimes, the Dalvik virtual machine (DVM) and the new Android runtime (ART), running on different system versions. Before Android 4.4, DVM is the runtime for executing the Dalvik bytecode in the Dex file. In Android 4.4, the system is equipped with both types of runtimes, DVM and ART, and the default runtime is DVM. From Android 5.0 released in 2015, the runtime of Android systems become ART. Fig. 1

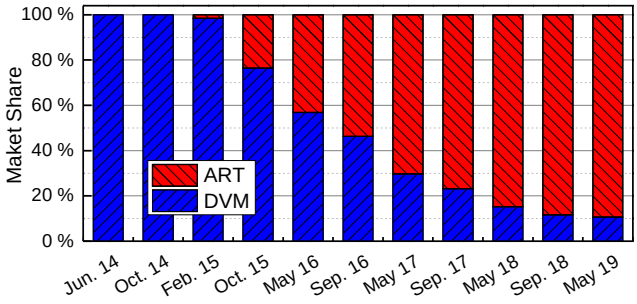

Fig. 1: The worldwide Android runtime share from Jun. 2014 to May 2019.

shows the worldwide Android runtime share from Jun. 2014 to May 2019. From this figure, we can find there are already around 90% Android devices running ART runtime in 2019.

2.3.1 The Dalvik Virtual Machine (DVM)

The DVM runtime adopts Just-In-Time (JIT) strategy to execute the Dalvik bytecode in the Dex files. The entire process from loading Dex files to the execution of bytecode can be divided into four major phases, including loading the Dex file, resolving the classes in the Dex file, resolving the methods in the Dex file, and executing the code of the methods. Specifically, when an Android app starts to run, the DVM runtime first loads its Dex files into the memory and stores the relevant information in the instances of `DexFile` structure. Thus each Dex file in the memory is referred by a corresponding `DexFile` instance. Afterward, the DVM runtime continues to resolve the classes as well as the methods implemented in the Dex file, and the resolving results are stored in the instances of `ClassObject` and `DexMethod` structures, respectively. Consequently, if a resolved method is invoked, its code will be executed by the DVM runtime. Note that, we also determine the basic Dex data collection points in DVM runtime according to this process and the details are presented in Section 4.1.

2.3.2 The New Android Runtime (ART)

By default, the ART uses the Ahead-Of-Time (AOT) approach to run the Android apps. Specifically, during the installation of an app, the ART runtime first compiles the Dalvik bytecode in its Dex files into native instructions by leveraging the tool `dex2oat` and then stores the compiling results as well as the original Dex files in a Oat file. In addition, if the Android system is upgraded, the Dex files of the apps need to be recompiled to the corresponding Oat files when the system upgrade finishes or the first time the device boots [56]. If an app contains no Oat file, the ART runtime also invokes `dex2oat` to generate the Oat file when it starts to run.

The ART runtime is also equipped with an interpreter for interpreting the Dalvik bytecode. Thus the ART runtime can execute a method in either the native mode or interpreter mode. In precise, by default, ART runs in native mode and executes compiled native code because this mode has better performance. However, if the method has no native code, ART interprets its Dalvik bytecode in the interpreter mode.

There are five major phases required for the ART runtime to execute the code of the methods, including loading the Oat file, parsing the Dex file, resolving the classes, resolving the methods, and executing the code of the methods. Precisely, when the target app starts, the ART runtime first loads its Oat file into the memory and uses an `OatFile` object to store the relevant information. Then the Dex file contained in the loaded Oat file is parsed by the runtime and the parsing results are stored in a `DexFile` object. Afterward, both the class and method information are resolved and the results are stored in the `Class` and `ArtMethod` objects. Thus the ART runtime can efficiently locate the code of a method and then execute it when the method is invoked. Such processing procedure can also help us to determine the basic Dex data collection points in ART runtime and the details are in Section 4.2.

2.4 Android App Packing

Intuitively, the Android packers usually protect the apps from three aspects, including hiding the Dex files, preventing Dex files from being dumped from memory, and improving the bar of static analysis.

2.4.1 Hiding Dex files

The original Dex files of the packed apps are usually hidden through dynamic releasing, dynamic modification, and reimplementing with native code. The major protection functionalities of the Android packers are implemented by native code, which protects the original Dex files in the packed apps through three major methodologies.

First, the protected Dex data is usually encrypted and stored in special files, and then the packers dynamically release them into the memory for execution during the running of the apps. Thus these protected Dex data cannot be statically obtained. It is worth noting that, for ART runtime, the unpackers can straightforwardly obtain the hidden Dex files if the protected Dex files are compiled by `dex2oat`. Consequently, to prevent the hidden Dex files from being obtained through such way, the packers usually dynamically releases the hidden Dex data and they leverage ART runtime to interpret the released bytecode.

Second, the packers also dynamically modify the Dex data in the memory to protect the completed protected Dex data being dumped from memory, and thus there may be specific Dex data missed in the dumped Dex data if the Dex data is not dumped at the correct Dex data collection points.

Third, we also find the sophisticated Android packers reimplement the functionalities of specific methods using native code. Hence there is no bytecode released for such protected methods throughout the running of the apps. Note that, although `PackerGrind` does not aim to convert the native code into bytecode, it still has the capacity of recovering the invoked framework APIs.

2.4.2 Preventing Dex Files from Being Dumped

The Android packers usually protect the Dex files from being dumped from memory directly from the following three major aspects.

First, the packers detect their running environments to prevent the packed apps from running on the emulator (i.e., Qemu) because much Android analysis and unpacking tools leverage Android emulator to carry out dynamic

```
1 private void initialize(void) {
2   if(Packer.getRunningEnv() == EMULATOR) // Check running environment
3      exit();                 // Exit if running on an emulator
4   Packer.antiMemDump(); // Prohibit memory dumping operations
5   Packer.antiDebug();   // Prohibit debugging Java or Native code
6   ...;                  // More initialization operations
7 }
```

(a) The initialization method

```
1 private void protectMethod(Bundle savedInstanceState) {
2 }
3 public void methodA(Bundle bundle) {// Protected methods
4   Int mthIndex = 0x89ab;            // The index of current protected method
5   Packer.releaseBytecode(mthIndex); // Release bytecode
6   protectMethod(bundle);          // The method containing original bytecode
7   Packer.destroyBytecode(mthIndex); // Destroy bytecode
8 }
```

(b) The method after being protected

Fig. 2: A motivating Android packer.

analysis [31], [72], [73]. If the packers detect they are running on the emulator, they will exit. To address this issue, `PackerGrind` runs on the real smartphone and carries out dynamic analysis by leveraging DBI (Dynamic Binary Instrumentation) framework [48].

Second, there are also multiple tools that reverse engineer and unpack Android apps using the debugging techniques (i.e., *ptrace()*) but the process can just be attached once for debugging. Hence, to prevent the packed apps from being debugged by the unpacking or analysis tools, the packers usually attach to the packed app with specific threads for anti-debugging. Note that `PackerGrind` does not rely on such techniques.

Third, the runtime unpackers usually dump the Dex data in memory through invoking the library functions of the Android system and thus the packers hook such functions to detect and prevent the invocations of these methods. Whereas, `PackerGrind` does not leverage these library functions to dump Dex data in memory.

2.4.3 Impeding Static Analysis

There are already various existing techniques widely used to protect Android apps statically, such as obfuscation [30]. The packers also leverage such techniques to impede the packed Dex file from being tampered straightforwardly when it is dumped by the unpackers. More precisely, the packers usually first obfuscate the code in the original Dex file and then pack the obfuscated Dex file during packing. When the packed app runs, the packers dynamically release the obfuscated code into memory for execution. As a result, after using unpackers to recover the obfuscated Dex file, the analysts need to further deobfuscate the bytecode in the recovered Dex file for the ease of comprehending the app. In particular, the obfuscation methodology aims to protect the target app by making it obscure to avoid tampering. Consequently, deobfuscation focuses on recovering the information from the obfuscated apps, such as implicit data and logics. Deobfuscation is out of the scope of this paper.

3 Motivating Example

Existing Android unpackers are implemented based on the knowledge through reverse engineering the Android packers, and perform a one-pass process to unpack the packed apps. Such unpackers can be easily circumvented by leveraging new or evolved packers because they are not adaptive to the changes of the packers. Consequently, the adaptive unpacking capacity is an active demand for Android unpackers.

We use the code snippets in Fig. 2 as a motivating example, which includes the initialization (i.e., Fig. 2a) implemented in the embedded packing Dex file that contains the packer's customized code, and a protected method (i.e., Fig. 2b) in the hidden/protected Dex file. As shown in Fig. 2a, when the packed app starts, the packer first checks its running environment, and it will exit if the emulator is detected (Line 2-3). Since some unpackers modify Android system to dump the Dex data in memory to storage or `ADB` (Android Debug Bridge) log. In Line 4 of Fig. 2a, the packer hooks the library functions related to data access and `ADB` log to prohibit dumping memory data. Moreover, it uses the anti-debugging techniques (Line 5) to prevent the bytecode and native code of the packed app from being debugged.

Fig. 2b shows the code of *methodA()* after it is packed. The original implementation of *methodA()* is replaced with the code between Line 4 and 7, as well as a method index (i.e., `0x89ab`) is assigned to it. Moreover, the callee *protectMethod()* is empty and invoked between two JNI functions (i.e., *Packer.releaseBytecode()* and *Packer.destroyBytecode()*) (Line 6). *Packer.releaseBytecode()* releases the original bytecode of *methodA()* to the code area of *protectMethod()* according to its parameter (i.e., the method index), and *Packer.destroyBytecode()* removes the bytecode of *protectMethod()*. Consequently, the original bytecode of *MethodA()* will be released to the code memory of *protectMethod()* by *Packer.releaseBytecode()* for execution, and then removed by *Packer.destroyBytecode()* after execution. Consequently, *protectMethod()* is released with various bytecode when it is invoked in different methods.

The existing unpackers, such as DexLego [50], DroidUnpacker [31], and TIRO [69], cannot effectively obtain the bytecode in *protectMethod()* (i.e., the original bytecode of *methodA()*). More precisely, DexLego and TIRO rely on the modified Android runtime and dump the Dex data in the memory. Since this packer prohibits memory dumping by hooking the related library functions (Line 4 in Fig. 2a), these two packers cannot handle this packer. DroidUnpacker runs on the emulator Qemu, and thus the packer will detect it and exit (Line 2-3 in Fig. 2a). In addition, existing unpackers usually collect Dalvik bytecode of a method once [71], [75], [78] but the packer can release different bytecode for a method at different call sites, and thus such unpackers cannot collect all the bytecode of these methods effectively.

`PackerGrind` can handle this packer because it runs on the real Android devices, iteratively monitors the packer's behaviors, and dumps memory data to files at multiple points (i.e., the invocations of Java methods, runtime functions, and library functions) leveraging the APIs of Valgrind [48] instead of library functions. Moreover, in addition to collecting the bytecode at special runtime functions, `PackerGrind` also gathers the bytecode when the Java method (i.e., *protectMethod()* in Fig. 2b) is invoked. Thus, it can collect all the bytecode released for *protectMethod()* when it is invoked at different call sites, and then reassembles all collected Dex data into a Dex file.

4 Basic Dex Data Collection Points

PackerGrind collects the Dex data at the basic data collection points by default, and will add more data collection points if more Dex data is needed after analyzing the behaviors of the target apps. Since all Dex data needs to be resolved by the runtime (i.e., DVM or ART) before being executed, we choose the basic Dex data collection points according to the entire process from loading a Dex file into memory to the execution of the bytecode in it. Such a process can be divided into multiple phases, including loading Oat files (only for ART), parsing Dex files, loading classes, resolving methods, and interpreting methods.

4.1 Basic Data Collection Points in DVM

In DVM runtime, the process from reading Dex file to executing the method can be divided into four major phases, including loading Dex file, loading classes, resolving methods, and executing methods, and thus we define four basic collection points to collect the dynamically released Dex data during unpacking.

4.1.1 Dex File Loading Point

During loading the Dex file of the app to run, the DVM runtime needs to parse the structures (i.e., the Dex file header) of target Dex file and thus we can collect the structure relevant information in the memory. Specifically, when loading a Dex file, the runtime uses method *dexFileParse()* to parse it and then store the parsing results in an instance of of structure `DexFile`. Then the `DexFile` instance is used to represent this Dex file in memory. Since `DexFile` is initialized according to the Dex file header in runtime method *dexFileParse()*, we locate the memory addresses of the loaded Dex file as well as the corresponding `DexFile` instance when *dexFileParse()* is invoked. Afterward, we can parse the Dex file in memory with the help of the corresponding `DexFile` instance to determine the addresses of specific Dex structures or items. Note that, we also reassemble the collected Dex data into valid Dex file based on the Dex structures represented by the `DexFile` instance.

4.1.2 Class Resolving Point

Since the packers can just release the class relevant information into memory before the classes are resolved, we collect Dex data when the classes are resolved by the runtime by default. During class resolving, the DVM runtime calls the method *dvmDefineClass()* to resolve the target class and stores the resolving result in a `ClassObject` instance, which contains the implementation details of the class, such as its properties, filed, and methods. Hence, we use the runtime method *dvmDefineClass()* as a basic Dex data collection point and collect the class relevant Dex data when the classes are resolved by the runtime.

4.1.3 Method Resolving Point

We also collect the Dex data when the methods in the Dex file are resolved by the DVM runtime because the packers can release the method relevant information only when they are used by the runtime for method resolving. More precisely, in memory, the DVM runtime represents each method using the `DexMethod` and the `Method` instances, which are initialized by the runtime methods *dexCompareNameDexcriptorAndMethod()* and *dexGetCode()* based on the method relevant information. The former is leveraged to check whether the method is a *finalize* one and the later is invoked to read the code related information from the Dex file. However, the method *dexGetCode()* is an inline function and not exported. Consequently, we specify *dexCompareNameDexDescriptorAndMethod()* as a basic point for locating the memory addresses of `DexMethod` as well as `Method` instances, and then collecting the method relevant information.

4.1.4 Method Execution Point

Since the bytecode of a method is just required when this method is interpreted by the DVM runtime concretely, the packers can release the bytecode only when the method is executed to prevent its code from being dumped. To address this issue, we choose to collect the bytecode of the methods when they are executed. Note that, the dynamically released methods can be invoked through two ways, JNI reflection or Java reflection through the runtime functions like *dvmInvokeMethod()*, *dvmCallMethodA()*, and *dvmCallMethodV()*. Since these runtime functions call *dvmInterpret()* for both fast-interpreter and portable-interpreter, we select it as the basic point to collect the bytecode of the methods when they are to be executed.

4.2 Basic Data Collection Points in ART

Similar to the processing phases in the DVM runtime, the ART runtime needs to interpret the dynamically released Dalvik bytecode through five phases, including loading Oat file, parsing Dex file, resolving classes, resolving methods, and executing methods. Hence we also specify the basic Dex data collection points in ART runtime according to these five phases.

4.2.1 Oat File Loading Point

Since the Dex file can be stored in the Oat file as a specific portion, the ART runtime first loads the Oat file into memory and then determines the memory region containing the Dex data through parsing the Oat file. Hence, we first locate the Dex file relevant information in the mapped Oat file when the Oat file is loaded into memory by the ART runtime. When an app starts to run, the runtime function *OatFile::Open()* is invoked to load and parse the target file (i.e., an Oat file). Also, a `OatFile` object is created to store the parsing result in the runtime function *OatFile::Setup()*. Hence, we locate the `OatFile` object in the memory as well as determine the memory addresses of the loaded Oat files when these two functions are invoked.

4.2.2 Dex File Parsing Point

Afterward, the ART runtime further parses the Dex files that are already mapped into the memory and hence we can collect the Dex file structure related information when the Dex files are parsed. In precise, the ART runtime invokes the methods *OatFile::OatDexFile::OpenDexFile()* and *DexFile::OpenFile()* to load the Dex files contained in the Oat file and stored in the storage into memory, respectively. Then each file in memory is referred by a `DexFile` object. Consequently, we leverage these two methods as the basic Dex data collection

points to locate the Dex files and `DexFile` objects in the memory. Such information will be used to collect the more Dex items in memory, such as `String_ids` and `Type_ids`, and reassemble the collected Dex data into valid Dex file.

4.2.3 Class Resolving Point

We collect the dynamically released class relevant Dex data (e.g., `class_def_item`) when the ART runtime resolves the classes in the Dex file. In ART runtime, each class is represented by a `Class` object that is created through loading and parsing the class related data in the Dex file by invoking the method *DefineClass()* of class `ClassLinker`. Consequently, we specify this function as a basic point for collecting the Class related Dex data in the Dex files as well as determining the `Class` objects in memory.

4.2.4 Method Resolving Point

We collect the Dex data related to the methods when the ART runtime resolves the method information (e.g., `registers_size` and `insns_size`) in the Dex file. The ART runtime uses the `ArtMethod` object to represent a method in the memory, which is initialized in the method *LoadMethod()* through parsing the method information. Then the initialized `ArtMethod` object is linked in the method *LinkCode()*. Hence we leverage these two methods to locate the `ArtMethod` objects and collect the corresponding implementation information.

4.2.5 Method Execution Point

We collect the code of the methods when they are interpreted by the ART runtime because the code is only actually used when the methods are executed. Specifically, *ExecuteSwitchImpl()* and *ExecuteGotoImpl()* are the entries of the two types of interpreters in ART runtime (i.e., the goto-based and switch-based interpreters). Thus we collect the bytecode stored in the `CodeItem` objects when the ART runtime starts to interpret the methods through both functions.

5 PackerGrind

In this section, we first present the overview of `PackerGrind` and then describe the details about the unpacking approach of `PackerGrind` including dynamic monitoring, Dex file recovery, as well as packing behavior analysis.

5.1 Overview

To be adaptive to various Android packers, `PackerGrind` adopts the learning-based unpacking process shown in Fig. 3 to recover Dex files. It includes two major phases: analyzing Android packers and unpacking apps.

5.1.1 Analysis of Android Packers

When a new packer is encountered, `PackerGrind` first leverages dynamic instrumentation mechanism (Section 5.2) to monitor the packers' behaviors in three layers (i.e., the runtime, system, and instruction layers) and generates a tracking report storing the execution trace of the packer. Meanwhile, `PackerGrind` dynamically collects Dex data at the specified points and then reassembles the collected data into a valid Dex file when the dynamic monitoring finishes (Section 5.3). Also, `PackerGrind` performs static analysis on

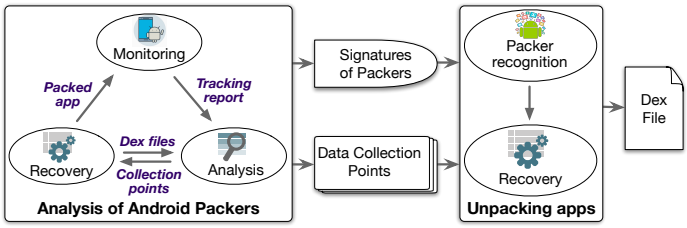

Fig. 3: The unpacking process realized by `PackerGrind`.

the Dex file to check whether the bytecode of all executed methods has been collected. If it is not true, more Dex data collection points are chosen through analyzing the tracking report and the Dex file (Section 5.4), and `PackerGrind` will re-run the apps to track more packing behaviors and collect Dex data at more collection points. Otherwise, a configuration file is generated to specify the Dex data collection points for `PackerGrind` to unpack the apps packed by this packer afterward.

Moreover, during the analysis phase, `PackerGrind` also generates the signatures of packers according to their initialization information. More precisely, the packers have different implementations and various protection code are added to target apps during packing. Thus we leverage three types of information as the packers' signatures, including the native libraries, classes, and methods, which are embedded into the packed apps by the packers to protect the original implementations (more details in Section 5.5.1). Note that, the signatures will be used to recognize the packers that pack the target apps during unpacking, which can speed up the unpacking process.

5.1.2 Unpacking Apps

Given an app to be unpacked, `PackerGrind` first determines its packer signature and then compares it with the signatures of the packers that have been analyzed (Section 5.5). If a new packer is encountered, `PackerGrind` will leverage the aforementioned adaptive mechanism to analyze this packer and determine the Dex data collection points. Otherwise, `PackerGrind` will directly collect the Dex data at the Dex data collection points determined for this packer and then reassemble the collected data into a valid Dex file.

Unpacking workflow: `PackerGrind` leverages an iterative process to unpacking a packed app through running it multiple times. Specifically, during the first running, `PackerGrind` runs the packed apps for a specified duration, during which `PackerGrind` first tries to trigger all the activities and services implemented in the app through generating various events. Meanwhile, `PackerGrind` collects the Dex data at the basic Dex data collection points (Section 4) and assembles them into a Dex file after the app stops. Note that, we identify the activities and services implemented in the app through decompiling the app and then parsing its manifest file. Also, we generate events by leveraging monkeyrunner [12] to trigger these activities and services.

Afterward, `PackerGrind` checks whether all methods implemented in the assembled Dex file contain bytecode. If so, the unpacking process finishes. Otherwise, more Dex data collection points are chosen through analyzing the tracking report and the assembled Dex file. Also, we determine which

events are required to trigger the invocations of the target methods and then instruct monkeyrunner to generate these events during the next running. Subsequently, `PackerGrind` runs the target app again. Such iterative unpacking process lasts until all methods in the assembled Dex file have bytecode or the number of successfully recovered methods does not increase. Note that, we regard the methods in the assembled Dex file with valid Dalvik bytecode as being recovered successfully because the packers can fill special methods with invalid data or dynamically released their valid bytecode during the running of the packed app. For example, when `PackerGrind` unpacks the app packed by the Qihoo packer, the iterative unpacking process terminates when the number of successfully recovered methods does not increase. Since Qihoo reimplements special methods with native code, the original contents of these methods are removed and never released. As a result, the contents of such reimplemented methods cannot be recovered even though the iterative process continues.

5.2 Monitoring

During dynamic monitoring, `PackerGrind` mainly focuses on tracking the Dex data protect relevant behaviors in multiple layers and collecting the Dex data at the default Dex data collection points.

5.2.1 Runtime Layer Monitoring

To track the dynamic releasing, modification and execution of the Dex data in ART runtime, we wrap 17 crucial ART runtime functions closely related to the protected Dex data (in Table 1). Then, `PackerGrind` can locate the memory of the Dex data according to parameters of the wrapped functions when they are invoked, and monitors the modification of the Dex data in memory. These functions can also be used as the data collection points to collect the dynamically released Dex data.

① *Dex Data Parsing and Loading.* As introduced in Section 4.2, when an app starts, the ART runtime invokes *OatFile::Open()* and *OatFile::Setup()* to open its Oat file and create a `OatFile` instance/object storing the basic information about this Oat file, such as the memory map of the Oat file and the Dex files contained in the Oat file. Then *OatFile::OatDexFile::OpenDexFile()* is utilized to parse the Dex files in the Oat file, which is already mapped into memory. Also, if the Dex file is dynamically loaded by the packer from the file directly, *DexFile::OpenFile()* is used to load the Dex file into memory. Both functions invoke *DexFile::OpenMemory()* to parse the Dex data in memory and represent each Dex file with a `DexFile` instance/object.

When the Oat file and Dex files are loaded and parsed, the ART runtime further employs *ClassLinker::DefineClass()* to resolve and link the loaded classes. Then, *ClassLinker::LoadMethod()* is utilized to load the methods, each of which will be represented as an `ArtMethod` instance/object, and then the code of the `ArtMethods` are linked in *ArtMethod::LinkCode()*.

Hence, by wrapping these functions, `PackerGrind` can identify the memory addresses of the target Oat and Dex data when these functions are invoked, such as the native code and bytecode of the methods represented by `ArtMethods`.

TABLE 1: The wrapped ART runtime functions and their corresponding target events.

Category	Wrapped Functions	Monitored Events
Dex Data parsing and loading	*OatFile::Open()* *OatFile::Setup()*	OatFile creation and setup
	OatFile::OatDexFile::OpenDexFile() *DexFile::OpenFile()* *DexFile::DexFile()* *DexFile::OpenMemory()*	Dex file's parsing and loading
	ClassLinker::DefineClass()	Class defining
	ArtMethod::LoadMethod() *ClassLinker::LinkCode()*	Method loading and linking
Method Execution	*InvokeWithVarArgs()* *InvokeWithJValues()* *InvokeVirtualOrInterfaceWithJValues()* *InvokeVirtualOrInterfaceWithVarArgs()*	JNI reflect invocation
	ExecuteGotoImpl() *ExecuteSwitchImpl()*	Method interpretation
Native Code	*JavaVMExt::LoadNativeLibrary()*	Native code loading
	ArtMethod::RegisterNative()	Native code reregistration

TABLE 2: Wrappers for tracking Dex and DVM related events.

Category	Wrapped Functions	Monitored Events
Dex Data	*dexFileParse()*	Dex file loading and parsing
	dvmClassDefine()	Class loading and defining
	dexCompareNameDescriptorAndMethod()	Method resolution
Method Invocation	*dvmInvokeMethod()*	Java reflection invocation
	dvmCallMethodV() *dvmCallMethodA()*	JNI reflection invocation
Native Code	*dvmLoadNativeCode()*	Native code loading

② *Method Execution.* Since the packer usually dynamically invokes the target methods through JNI reflection, `PackerGrind` monitors such behaviors through wrapping four JNI related functions, *InvokeWithVarArgs()*, *InvokeWithJValues()*, *InvokVirtualOrInterfaceWithJValues()*, and *InvokeVirtualOrInterfaceWithVarArgs()*, which are utilized to invoke methods according to the types of arguments. `PackerGrind` monitors the executions of Dalvik bytecode because the packer could release the protected bytecode just before execution and then erase them after execution. In ART runtime, two types of interpreters are implemented, the goto-based interpreter and the switch-based interpreter. Correspondingly, the functions *ExecuteGotoImpl()* and *ExecuteSwitchImpl()* are used to call the specific interpreter, respectively. Hence, `PackerGrind` wraps these two functions to monitor the interpretation of methods' bytecode. These two functions are also used to track the methods with only bytecode in Section 5.2.4.

③ *Native Code Loading and Registration.* The Android packers usually implement their major protection mechanisms in native code and provide JNI method for invoking the native code. Thus, when a packed app starts, the runtime first loads the native library of the Android packer in *JavaVMExt::LoadNativeLibrary()*, and then registers the `ArtMethods` of JNI methods with their corresponding native code in *ArtMethod::RegisterNative()*. Consequently, `PackerGrind` wraps *JavaVMExt::LoadNativeLibrary()* to locate the native code of the packers and *ArtMethod::RegisterNative()* to locate the na-

tive code of every JNI `ArtMethods` for tracking the behaviors of the packers.

5.2.2 System Layer

To protect the protected Dex data (e.g., the original Dex file or a specific Dex item) from being dumped, the packers usually invoke system library functions to dynamically release and modify the Dex data in memory and such functionalities are implemented in their native modules. Hence, to track such behaviors, `PackerGrind` monitors the invocations of the corresponding library functions through wrapping them in the system layer. Currently, `PackerGrind` supports various types of functions, including memory management functions (e.g., *alloc()* and *free()*), file operations (e.g., *read(), write(), and* open(), and data movement functions (e.g., *memcpy()* and *strcpy()*). Additionally, `PackerGrind` also tracks the invocations of specific system calls (e.g., *sys_protect(),* sys_map(), *and* sys_unmap()) which are usually leveraged by the packers to dynamically release and modify the hidden Dex data, such as allocating memory regions and the modifying memory access permissions. Thus, through monitoring these system calls, `PackerGrind` can determine the memory regions storing the Dex data and then monitor the modifications of these Dex data.

Moreover, `PackerGrind` can also be leveraged to perform other special actions through wrapping the corresponding functions. For instance, to detect whether the packed apps are being analyzed, there are packers checking the time consumed by special operations (e.g., iterative loop), such as the Ijiami packer. Hence, to avoid being detected, we specify `PackerGrind` to change the results of *sys_gettimeofday()* through wrapping this function because it is invoked to obtain the timestamps for time checking.

5.2.3 Instruction layer

Besides invoking the data copy and move functions (e.g., *memcpy() strcpy()*), the packers can also modify the Dex data in memory leveraging their customized code and hence `PackerGrind` dynamic tracks the *Store* instructions for monitor Dex data operations. Since `PackerGrind` has identified the memory locations of both the packing libraries and the system libraries, it can determine whether the *Store* instructions belong to the packer's libraries according to their addresses.

Specifically, `PackerGrind` dynamically inserts tracking statements before each *Store* (i.e., `Ist_StoreG` and `Ist_Store`) statement when the native code is translated into IR statements and then the tracking statements will be invoked just before the *Store* statements. Since `PackerGrind` maintains all the memory regions storing the released Dex data, it first checks whether the Dex data is modified according to the target addresses of the tracked *Store* statements. If so, the information about the statement addresses, the target addresses, and the data to be stored are all logged into the tracking reports for further Dex file reassembling and new Dex data collection point determination.

5.2.4 Java Method Invocation Tracking

To collect the dynamically released Dalvik bytecode at the method level, `PackerGrind` also tracks the invocations of the Java methods. Consequently, even if different implementations/bytecode are released to the same Java method under protection at different call sites, `PackerGrind` can collect all the released bytecode when they are invoked. Generally, the methods to be tracked by `PackerGrind` can be divided into three major types, the compiled methods with both Dalvik bytecode and native code, the JNI methods that have native code in customized libraries without Dalvik bytecode, and the protected methods containing dynamically released bytecode and no native code. They are tracked using different mechanisms.

① *The Compiled Methods.* `PackerGrind` first parses the OAT files, which are generated from the embedded packing Dex files by `dex2oat`, to obtain the entry points of the compiled methods according to the beginning addresses of their native instructions. When the native instructions resulted from the Dalvik bytecode are translated to IR blocks by valgrind, `PackerGrind` dynamically inserts additions statements (i.e., the callback function *getCallee()*) at the method entry points. Note that an IR block is a collection of IR statements with one entry and multiple exit points. Consequently, if the entry point of an IR block is also the entry point of a method, `PackerGrind` inserts an additional IR statement to call *getCallee()* to identify the invoked method and parse its parameters.

However, it is not straightforward to monitor the exiting of the compiled methods, because a method usually contains multiple exit points and returns to different addresses in its different call sites. `PackerGrind` leverages the method call convention of ARM architecture to overcome this challenge. Precisely, we design a customized method call stack structure `mthStack` in `PackerGrind`. Thus, in *getCallee()*, `PackerGrind` reads the return address of the method to be invoked from the link register `lr` and pushes it into `mthStack`. `PackerGrind` inserts additional IR statement before each branch (i.e., `Ist_Exit` and `Ist_Next`) statement to call the callback function *isMthRet()* for checking whether the current method is returning. In *isMthRet()*, `PackerGrind` compares the target address of the branch statement with the top address in `mthStack`. If they are equal, the current function will return, and the `PackerGrind` pops the top address from `mthStack`. Also, `PackerGrind` parses the method result in *isMthRet()* if the method returns the result.

② *The JNI Methods.* We adopt a similar mechanism to track the JNI methods. However, the native code of the JNI methods are implemented in the customized native libraries, which are also protected by the packers. Moreover, the native code is dynamically registered to the `ArtMethod` objects. Note that, in ART runtime, each method is represented by an `ArtMethod` object, which stores its general information (e.g., the offset of its Dalvik code item and the method index in the Dex file). Hence it is unable to identify the memory addresses of their native code through parsing the `ArtMethod`s or decompiling the native libraries in advance. To address this issue, `PackerGrind` wraps the runtime function *ArtMethod::RegisterNative()* because it is used to register the native code to the corresponding `ArtMethod` object. Thus `PackerGrind` obtains the memory addresses of the JNI methods when this function in invoked.

③ *The Interpreted Methods.* To protect special Dalvik bytecode against being unpacked when they are compiled into native

instructions by dex2oat, the packers usually release the protected bytecode into memory during execution, and then these bytecode will be interpreted by the ART runtime. Note that, the packers may release different implementations for the same method at different call sites. Hence, we need to track the interpreted methods for collecting all their dynamically released bytecode.

We choose to monitor the interpreter of ART runtime for tracking the invocations of the methods without native code. Precisely, for executing the Dalvik bytecode, there are two types of interpreters implemented in ART runtime: the goto-based interpreter and the switch-based interpreter, which dispatches the Dalvik bytecode instructions through `goto` and `switch`, respectively. They provide the functions *ExecuteGotoImpl()* and *ExecuteGotoImpl()* for the runtime to interpret a method, respectively. Note that, one parameter of these two functions is the address of the bytecode.

Consequently, to track the interpreted methods, `PackerGrind` identifies the memory of these methods' bytecode when they are linked by the runtime function *ClassLinker::LinkCode()*. Note that, `PackerGrind` only obtains the memory addresses of the methods' bytecode instead of the bytecode, and the packer could dynamically release the valid bytecode just before the methods being interpreted. Hence, `PackerGrind` wraps *ExecuteGotoImpl()* and *ExecuteGotoImpl()* to identify the memory address of the bytecode to be interpreted, and determines the invoked methods according to these addresses.

5.3 Recovery

To recover valid Dex files from the packed apps, `PackerGrind` first collects the dynamically released Dex data in the memory and then reassembles them into Dex files, which can be directly decompiled by the existing static analysis tools [6], [8]–[11].

5.3.1 Dex data collection

When the packed app is launched, `PackerGrind` first locates the `DexFile` object representing the Dex file in the memory, and then starts to collect the Dex data through parsing this object. When `PackerGrind` finds a new `DexFile` object, it will initialize a shadow `DexFile` to store the collected Dex data because the Dex data represented by the original `DexFile` object may be dynamically modified by the packer. Afterwards, when a new Dex data item is collected at the Dex data collection points, `PackerGrind` first checks whether the data item is already in the memory represented by the shadow `DexFile`. If so, `PackerGrind` directly copies the collected data to the target memory. Otherwise, `PackerGrind` first creates a new memory structure and then copies the data to this memory structure. Also, `PackerGrind` set the corresponding offsets of the shadow `DexFile` and then this new collected data item can be located through parsing `DexFile` object.

5.3.2 Dex file assembling

Since the packer usually releases the hidden Dex data into discontinuous memory regions to prevent the Dex data from being directly dumped, the collected Dex data in the shadow memory are not continuous and we need to reassemble them into a valid Dex file just before exiting. Consequently, `PackerGrind` achieves this purpose through two major steps. First, `PackerGrind` puts the collected Dex data items of the same type together because the data structures storing the same type of data are stored in the continuous memory regions in the Dex file. Second, `PackerGrind` allocates a continuous memory region for storing the new Dex file and then copies all reconstructed Dex data structures to it. Meanwhile, `PackerGrind` sets the size and offset fields of these structures as well as other metadata. Finally, the structured and linkable data in the continuous memory region is dumped to the storage as the reassembled Dex file.

Recovering method with various implementations: Such as the motivating example in Section 3, different implementations could be dynamically released to the same method at different call sites. To address this issue, `PackerGrind` collects different bytecode released to a method at different call sites, and adds a new method containing the newly collected bytecode to replace the original method at this call site. For example, Fig. 4 shows the unpacking results of the motivating example (i.e., Fig. 2b) and the method *protectMethod()* is replaced with the newly added method *methodA_protectMethod_1()* that contains the bytecode collected at the call site in Line 7 of Fig. 4.

Native methods inspection: The bytecode of the specific Java methods in the apps can be re-implemented leveraging native code during being packed, and such methods are then invoked through JNI during the running of the packed apps.

```
1 private void methodA_protectMethod_1(Bundle savedInstanceState) {
2   ...; // Code recovered from the packed app during runing
3 }
4 public void methodA(Bundle bundle) {
5   Int mthIndex = 0x89ab;
6   Packer.releaseBytecode(mthIndex);
7   methodA_protectMethod_1(bundle);  // Replaced with the new unapcked method
8   Packer.destroyBytecode(mthIndex);
9 }
```

Fig. 4: The unpacking results of the motivating example.

```
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.*

```
1 Invoke:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILLL)
2    JNI_Reflection: Landroid/app/Activity; onCreate(VL)
3    JNI_Reflection: Landroid/app/Activity; setContentView(VI)
4    JNI_Reflection: Landroid/app/Activity; getSystemService(LL)
5    JNI_Reflection: Landroid/view/WindowManagerImpl; getDefaultDisplay(L)
6    JNI_Reflection: Landroid/gesture/GestureLibraries; fromRawResource(LLI)
7    JNI_Reflection: Landroid/gesture/GestureLibraries$ResourceGestureLibrary; load(Z)
8    JNI_Reflection: Landroid/app/Activity; findViewById(LI)
9    JNI_Reflection: Landroid/gesture/GestureOverlayView; addOnGesturePerformedListener(VL)
10 Return:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILLL)
```

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

Fig. 5: The method *onCreate()* invoked before and after packing and tracked information about *A.V()*.

Although PackerGrind is not designed to convert the native code into bytecode, it can provide useful information related to the invocations of the native methods because of its cross-layer monitoring capability. More precisely, packed apps have to use JNI reflection to invoke Android framework APIs in Java. Since PackerGrind tracks both the app's customized Java methods and the Android framework methods, it can provide rich information about the interactions between the native code and the Java code.

For example, Fig. 5 shows the method *onCreate()* in the original app (Fig. 5a) and that in the app packed by Baidu (Fig. 5b). After being packed, the functionalities of *onCreate()* have been re-implemented in the native code of JNI method *A.V()* and it leverages JNI reflection to call the original Android framework APIs. From the tracked callees of *A.V()* shown in Fig. 5c, we can infer the original implementation and functionalities of *onCreate()*. Take the method *onCreate()* of class *android/app/Activity* invoked at Line 2 of Fig. 5c as an example, it corresponds to the method invocation at Line 2 of Fig. 5a. Also, according to the API *addOnGesturePerformedListener()* invoked at Line 9 in Fig. 5a, we can infer that one functionality of method *onCreate* is to add a gesture listener to the activity [20]. Consequently, the tracked callees can also assist the framework API based static analysis tools (e.g., PScout [22] and DroidSafe [35]). Note that, other unpackers (e.g., [50], [69], [75], [78]) cannot profile such behavior.

5.4 Analysis

After the dynamic monitoring of a new packer, a tracking report is generated containing the information about the invoked customized Java methods, the Android framework APIs, the specified runtime functions, the system functions, and the modifications of special memory (i.e., the Dex data related regions). PackerGrind identifies the behaviors of the packer through analyzing the tracking report, and then generates the packer's signature and determines the Dex data collection points according to its behaviors.

During dynamic monitoring, PackerGrind focuses on three types of information, which are also stored in the tracking report for further analysis.

• *Dex data related items.* If a DexFile object that represents a new Dex file in the memory is found, PackerGrind will obtain the information related to the metadata, classes, methods and bytecode in the Dex file and then output them to the tracking report. Moreover, the object ArtMethod is leveraged to represent a Java method in the runtime, and hence PackerGrind also logs the information related tho ArtMethod to the tracking report for analyzing the invoked methods.

• *Memory modification.* PackerGrind maintains a Dex file list containing the memory ranges of all Dex related data in the runtime, such as DexFile objects and ArtMethod objects. When functions or instructions are used to modify the data in these memory, PackerGrind writes the modification information into the tracking report. Note that, the packers can release the data into memory through two ways. One is to release the valid data into the memory of the Dex file directly, and the other is to modify the corresponding pointer to the memory addresses that contain the valid bytecode. The system layer information is related to the invoked functions, target addresses, the Dex data structures containing the target address (e.g., Dex header field), and data written to the target addresses. Additionally, the instruction layer information is related to the instruction addresses, instruction types (i.e., Ist_StoreG and Ist_Store), target addresses, and the stored value.

• *Method invocation.* PackerGrind monitors both the invocations and the returns of the wrapped methods as well as their parameters and results in the runtime layer and the system layer. Note that, the monitored functions include both JNI/compiled functions and the Java methods and all the monitored information are logged into the tracking reports.

The behaviors of Android packers can be roughly divided into two types, including the initialization behaviors and the dynamic Dex data releasing behaviors, which will be leveraged to recognize the packers and determine Dex data collection points, respectively.

5.4.1 The Initialization Behaviors

The code of a packed app includes both the code of the Android packer (i.e., the bytecode of the embedded packing Dex file) and the original protected code (i.e., the bytecode of the original Dex files). The embedded packing code first execute to load the native code and initialize the execution environment before releasing the protected Dex files. Since the initialization behaviors are completely performed by the packers, they are packer-specific and we recognize packers based on such behaviors (details in Section 5.5).

5.4.2 Dex Data Releasing Behaviors

PackerGrind analyzes the Dex data releasing behaviors with the purpose to identify the points when the valid Dex data are released into the memory by the packers. During analyzing the Dex data releasing behaviors, PackerGrind focuses on the memory data modifications of both the data items in the Dex file and the data objects that are used to represent the Dex data in runtime, such as the DexFile and ArtMethod objects, because the packers need to guarantee that the Dex data is valid when it is executed or interpreted by the runtime.

By exploiting the insight that a tracked portion (i.e., an item in the Dex file or a Dex related object) should be valid right before being used, we summarize four major types of protection patterns: (1) it is always valid (**T**); (2) it is changed to valid value before its first use and not modified afterward (**FmT**); (3) it is valid until its last use and then modified to invalid value (**TmF**); (4) it is altered to valid value before being used and turned to invalid after the use (**FmTmF**); Although the basic protection patterns are by no means comprehensive, they cover all packed samples available to us. Note that PackerGrind also supports users to define customized patterns through analyzing the tracking report.

PackerGrind identifies the protection pattern of each Dex data item and object (i.e., P) through analyzing the modification on it and its use in the tracking report. Note that PackerGrind determines the use of P according to the invoked runtime functions and Java methods. For example, when a class is defined in *DefineClass()*, the corresponding class data item in the Dex file is valid. Similarly, the bytecode of a method is valid when the method is interpreted in

ExecuteSwitchImpl() or *ExecuteGotoImpl()*. After the analysis of the Dex data releasing behaviors, PackerGrind outputs the modifications of each Dex data item P as well as its status (i.e., valid or invalid) according to the corresponding invoked methods, and this information will be used to determine the Dex data collection points in Section 5.4.3.

5.4.3 Dex Data Collection Point Determining

PackerGrind can specify the invocations or returns of the special methods as data collection points and it determines such points according to the packing behaviors of the packers. For the Dex data items protected through **T** and **TmF**, PackerGrind collects the valid data when they are first encountered in the memory. If the Dex data items are protected using **FmT** and **FmTmF**, PackerGrind collects their data when they are changed to valid values.

Furthermore, PackerGrind compares all the bytecode collected for a Java method to determine whether the method has various implementations. If it is true, PackerGrind will perform method granularity bytecode collection for this method, namely, PackerGrind will track the invocations of this method and collect its bytecode every time it is invoked.

For a newly encountered packers, since no packing behaviors tracked, PackerGrind leverages the default Dex data collection points to dump the Dex data in memory during the first run, and thus the recovered Dex file may contain invalid data. For such packers, new Dex data collection points will be determined during the analysis phase and then we run PackerGrind again to collect the valid Dex data at points.

5.4.4 Static Bytecode Analysis

Since the dynamic Dex data releasing and modification functionalities of the packers are always implemented in the native code of special JNI protection methods, it is a challenge to trigger these methods to release the protected Dex data.

Currently, we leverage IntelliDroid [68] to determine how to trigger these JNI methods through extracting the call graphs from the Dex data that is already dumped. Precisely, given a set of targeted behaviors (e.g., specific framework APIs), IntelliDroid traverses the app's call graph to find paths to these behaviors. Meanwhile, it also extracts path constraints, which are used to determine the input values that can trigger these paths. Then, IntelliDroid takes extracted paths/constraints and injects the generated input values to the Android device to trigger the targeted behaviors. Consequently, we first determine the JNI methods in the Dex file that is already unpacked from the target app, and then leverage IntelliDroid to identify the execution paths to these functions. Afterwards, we trigger the app to execute the identified paths and invoke these JNI methods to release the target protected bytecode for collection. Such as the motivating code shown in Fig. 2b, the real bytecode of method *protectMethod* is just released into memory when the JNI protection method *Packer.releaseBytecode* is invoked. Consequently, the method *protectMethod* dumped before invoking *Packer.releaseBytecode* contain no bytecode, and thus we further collect its bytecode through triggering *Packer.releaseBytecode* by applying IntelliDroid to identifying the execution path to it.

```
"Ali": {
  "clazz": [
    "Lcom/ali/mobisecenhance/StubApplication;"
  ],
  "library": [
    "libmobisec.so",
    "libmobisecx.so",
    "libmobisecy.so",
    "libmobisecz.so",
    "libmobisecz1.so",
    "libmobisecy1.so"
  ],
  "method": [
    "attachBaseContextIT"
  ],
  "size": 8
}
```

Fig. 6: The signature of Ali packer.

5.5 Packer Recognition

To recognize the packers that pack the target apps, we generate the signatures of the known packers through extracting the packing features of apps packed by them previously (Section 5.5.2). Thus, during unpacking phase, we also extract the packing features of the target packed apps (Section 5.5.1) and then recognize the packers through comparing the extracted features with the signatures of known packers (Section 5.5.3).

5.5.1 Extracting Packing Features of Apps

Although the apps have various implementations, similar code will be embedded to protect the original bytecode, such as detecting running environment and releasing protected bytecode, when they are packed by the same packer. Hence, we extract three types of embedded information from each packed app as its packing features, including the native libraries ($\mathtt{f}_{\mathtt{lib}}$), classes ($\mathtt{f}_{\mathtt{cla}}$), and method ($\mathtt{f}_{\mathtt{mth}}$) embedded by the packer. In particular, the classes and methods are usually used to detect and initialize the running environment. The code in the native libraries usually focus on the functionalities that cannot be realized by bytecode, such as dynamically releasing and protecting the original Dex data during the running of the app, because bytecode cannot directly operate memory and system (e.g., allocating memory, hooking functions, and injecting processes). Given a packed app a, PackerGrind first extracts its packing features before collecting the protected Dex data and we use $\mathtt{F}_{\mathtt{a}}$ to denote the packing features of the packed app a, namely $\mathtt{F}_{\mathtt{a}} = \mathtt{f}^{\mathtt{a}}_{\mathtt{lib}} \cup \mathtt{f}^{\mathtt{a}}_{\mathtt{cla}} \cup \mathtt{f}^{\mathtt{a}}_{\mathtt{mth}}$.

5.5.2 Building Signatures of Packers

Since a packer usually has various versions but perform similar packing behaviors, we combine the packing features extracted from all the known apps packed by the packer P (details in Section 5.5.1) as the signature ($\mathtt{S}_{\mathtt{P}}$) of this packer. In this paper, we use P to represent a packer that has multiple versions and $\mathtt{S}_{\mathtt{P}}$ to denote its signature. The $\mathtt{S}_{\mathtt{P}}$ contains three types of information, including the native libraries $\mathtt{s}^{\mathtt{P}}_{\mathtt{lib}}$, classes $\mathtt{s}^{\mathtt{P}}_{\mathtt{cla}}$, and methods $\mathtt{s}^{\mathtt{P}}_{\mathtt{mth}}$ embedded into the apps packed by the packer of various versions. In addition, the signatures of the packers are updated during unpacking

apps. More specifically, if an app $\mathtt{a}'$ packed by the unknown packer $\mathtt{P}'$ is encountered, we identify $\mathtt{P}'$ manually and then add the packing features (i.e., $\mathtt{f}^{\mathtt{a}'}_{\mathtt{lib}}$, $\mathtt{f}^{\mathtt{a}'}_{\mathtt{cla}}$, and $\mathtt{f}^{\mathtt{a}'}_{\mathtt{mth}}$) of $\mathtt{a}'$ to the signature of $\mathtt{P}'$ (i.e., $\mathtt{s}^{\mathtt{P}'}_{\mathtt{lib}}$, $\mathtt{s}^{\mathtt{P}'}_{\mathtt{clz}}$, and $\mathtt{s}^{\mathtt{P}'}_{\mathtt{mth}}$).

For example, Fig. 6 shows the signatures built for Ali packer, which includes 8 items, namely one native library, six classes, and one method, and this signature is generated based on the apps packed by three different versions of Ali packer. It is worth noting that, to guarantee that the signatures of different packers just contain the unique items, we remove the items from signatures if they exist in the signatures of multiple packers. Consequently, the signatures of various packers do not share same item, namely $\mathtt{S}_{\mathtt{P}'} \cap \mathtt{S}_{\mathtt{P}} = \emptyset$ if $\mathtt{P}'$ is not $\mathtt{P}$.

5.5.3 Recognizing Packers

We also recognize the packer of a newly encountered packed app according to its packing features $\mathtt{F}_{\mathtt{a}}$ extracted in Section 5.5.1. Specifically, we calculate the similarity (i.e, $\mathtt{Similarity}(\mathtt{a}, \mathtt{P})$) between the packing features $\mathtt{F}_{\mathtt{a}}$ of $\mathtt{a}$ and the signature $\mathtt{S}_{\mathtt{P}}$ of each known packer $\mathtt{P}$ as follows:

$$\mathtt{Similarity}(\mathtt{a}, \mathtt{P}) = \frac{|\mathtt{F}_{\mathtt{a}} \cap \mathtt{S}_{\mathtt{P}}|}{|\mathtt{F}_{\mathtt{a}}|} \quad (1)$$

We regard the target app $\mathtt{a}$ as being packed by the packer $\mathtt{P}$ if the similarity between $\mathtt{F}_{\mathtt{a}}$ and $\mathtt{S}_{\mathtt{P}}$ is larger than zero. Note that, since $\mathtt{S}_{\mathtt{P}}$ just contains the unique items that are not shared by various packers (Section 5.5.2), $\mathtt{F}_{\mathtt{a}}$ and $\mathtt{S}_{\mathtt{P}}$ share same item only when the app $\mathtt{a}$ is packed by $\mathtt{S}_{\mathtt{P}}$, namely $\mathtt{Similarity}(\mathtt{a}, \mathtt{P}) > 0$. If the similarities between $\mathtt{F}_{\mathtt{a}}$ and the signatures of all know packers are zero, we regard $\mathtt{a}$ as being packed an unknown packer, and then leverage the approach described in Section 5.3 to unpack it. Otherwise, we start to unpack the target app through collecting the Dex data at the points that are already determined for this packer.

6 Evaluation

We have re-implemented PackerGrind with a focus on ART runtime (Android 6.0) as well as the enhanced and new functionalities are described in this paper. Specifically, the current version of PackerGrind is implemented with 13453 and 12894 lines of C/C++ code for the DVM and the ART, respectively. Also, we implement Python scripts to automatically unlock phones and run the target apps with 1896 lines of code. The dynamical tracking and unpacking subsystem of PackerGrind leverages Valgrind, the dynamic binary instrumentation framework [48], to generate the tracking reports and collect Dex data. We implement the tracking report analysis subsystem in Python for generating the packers' signatures and determining the Dex data collection points through analyzing the tracking reports.

In this section, we will evaluate the capacity and performance of PackerGrind through answering the following questions.

- **RQ1:** Can PackerGrind identify the protection mechanisms of the packers and determine the Dex data collection points?
- **RQ2:** Can PackerGrind correctly recognize the packer and recover the Dex files?
- **RQ3:** Can PackerGrind outperform other available unpackers?
- **RQ4:** Can PackerGrind facilitate the static malware detection tools?
- **RQ5:** Is overhead of PackerGrind acceptable?

6.1 Data Set

In this section, two sets of samples that are packed by multiple popular commercial packers are leveraged to evaluate PackerGrind. The first set has 680 packed apps with ground truth. More precisely, we randomly downloaded open-source apps from F-Droid [5] and then upload them to six online commercial packing services (i.e., Qihoo [54], Ali [19], Bangcle [25], Tencent [64], Baidu [24], Ijiami [37]) in Mar. 2015 (denoted as DB-15), Mar. 2016 (denoted as DB-16), and Oct. 2018 (denoted as DB-18). Correspondingly, 680 packed apps are downloaded from these packing services. Note that, since Ali packer did not provide online packing services in Oct. 2018, DB-18 does not contain any app packed by Ali.

The second set consists of 399 packed malware samples from Palo Alto Networks [51], of which 389 samples were packed by 12 known packers, including Ali [19], APKProtect [1], Baidu [24], Bangcle [25], Ijiami [37], Naga [47], Qihoo [54], Tencent [64], LIAPP [60], Netqin [49], Payegis [52], and NetEase [38], and the remaining malware are packed by unknown packers.

We conduct the experiments for both Android 4.4 (i.e., AOSP build KTU84M) and Android 6.0 (i.e., AOSP build MMB29V) running with DVM and ART runtime, respectively, and all the experiments are carried out on the Nexus 5 smartphone. For each packer, we first leverage PackerGrind to identify its protection patterns and then recover the protected Dex files in its packed apps.

6.2 Analysis of Android Packers

We first leverage PackerGrind to reveal the protection mechanisms adopted by the six packers, each of which has three versions for DB-15, DB-16, and DB-18 (except Ali), respectively. As shown in Table 3, the packers have evolved with new techniques and hence unpackers should be adaptive to the evolution. Except for the methods re-implemented using native code, PackerGrind can unpack all the bytecode protected by various mechanisms in the packed apps.

6.2.1 Protection Mechanisms

As shown in Table 3, except the apps packed by Tencent in DB-15, the Dex files in other packed apps are all dynamically released into memory during running. Also, for the packed apps in DB-16, all packers but Baidu dynamically modify specific Dex data structures in memory to prevent the protected Dex file from being directly dumped. Such as the Ali packer, it dynamically fills the `class_data_item` structures with valid data just before the corresponding classes are defined. Similarly, for the apps packed by Ijiami packer, the Dex headers in their Dex files are filled with valid data when *dexFileParse()* is invoked to parse the header and then set with invalid values when *dexFileParse()* exits.

TABLE 3: Protection mechanisms adopted by six packers in DB-15, DB-16 and DB-18. The symbol before (or after) "—" denotes whether a packer in DB-15, DB-16, or DB-18 uses the mechanism or not.

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

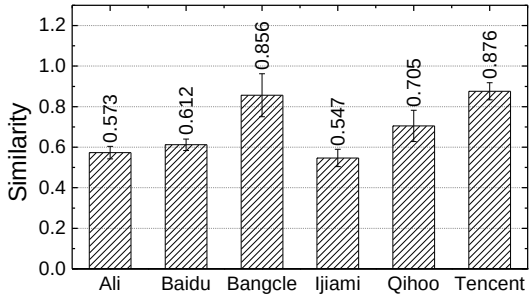

Fig. 7: The similarities between each packer's signature and the packing features of the app packed by its different versions.

In DB-16, there are also packers invoking their customized code to parsing special Dex data structures instead of calling the runtime functions, such as Qihoo and Ijiami. Specifically, Qihoo re-implements the class define functionality in its native library libjiagu.so and thus these code are dynamically invoked to define the classes in the Dex files instead of *dvmDefineClass()*. Ijiami also implements special method parsing functionalities in its native library libexec.so for re-parsing methods of the loaded classes with valid instruction offsets before the classes are defined. In addition, there are also packers re-implementing special methods in the protected Dex files with native code. For example, in DB-16, the *onCreate()* methods of the apps packed by Baidu are re-implemented using native code and then the original bytecode of these methods are never released into memory during the entire running process. In addition, to prevent the packed apps from being analyzed by *ptrace*-based debugging tools, Bangcle attaches the app process by itself through invoking *ptrace()*. Moreover, Ijiami periodically detects the ZjDroid framework through searching the string "*@com.android.reverse-*" in memory [14].

In either DB-15 or DB-16, only Baidu re-implements the specified methods (e.g., *onCreate()*) in the native library using C/C++ code. In addition, Qihoo also adopts this mechanism to protect the important Java methods in DB-18. However, both packers only re-implement special methods in the native code. One possible reason may be that re-implementing bytecode in native code could introduce additional overhead (e.g., the overhead due to context switch).

6.2.2 Dex Data Collection Points.

We use the latest packers in DB-18 to evaluate `PackerGrind` in terms of determining Dex data collection points. Qihoo and Baidu re-implement the special methods using native node, but other items in the Dex files are dynamically released to memory with valid values before the invocation of the runtime function *DexFile::OpenMemory()*. Therefore, `PackerGrind` collects the valid Dex data of the apps packed by these two packers when *DexFile::OpenMemory()* is invoked. Similarly, Bangcle and Tencent release all Dex items with valid values before invoking *DexFile::OpenMemory()* and thus `PackerGrind` collects the valid Dex data when *DexFile::OpenMemory()* is invoked. However, Ijiami first releases the Dex data with invalid `CodeItem` before invoking *DexFile::OpenMemory()* and then changes the `CodeItems` to valid values during invocations of *ArtMethod::LoadMethod()*. Therefore, when *DexFile::OpenMemory()* is invoked, `PackerGrind` collects the Dex data items except `CodeItems`, and then collects the valid `CodeItems` when *ArtMethod::LoadMethod()* returns.

Answer to RQ1: `PackerGrind` can successfully identify the protection mechanisms of the packers and determine the Dex data collection points for unpacking.

6.3 Recovering Dex Files

In this experiment, we will evaluate whether `PackerGrind` can effectively recognize the packers of the packed apps and then correctly recoverer the protected Dex files of these packed apps by leveraging the apps packed by various commercial packers.

6.3.1 Packer recognition

In this experiment, we select an app from the apps packed by each packer in the three datasets for generating packer signatures, and then using the remaining packed apps together for recognition. More precisely, in the analysis phase, we generate the signatures of all these packers using the mechanism introduced in Section 5.4, and then recognize the packers that harden these apps in the unpacking phase. The experimental results show that `PackerGrind` correctly recognizes all these six packers. Moreover, the similarities between the packers' signature and the app's packing features are presented in Fig. 7. For all these packers, the similarities are larger than 0.5. In addition, since the packer signatures just contain the unique feature of each packer, the similarities between the signature of a packer and the packing features of the apps packed by different packers are all zero.

6.3.2 Correctness of recovered Dex files

The correctness of the Dex files recovered from the packed app are examined from three major aspects. First, we apply five popular static analysis tools, including `Baksmali` [6], `Dexdump` [9], `Dex2jar` [8], `Jadx` [11] and `IDA Pro` [10], to disassemble the recovered Dex files because these tools adopt different verification strategies to check Dex files. From the results, we find almost all the recovered Dex files are successfully disassembled, and the only samples packed by Tencent in DB-15 are disassembled by `Dex2Jar` with exceptions because of their optimized Dex bytecode.

TABLE 4: Difference between the code in the Dex file of the original app and the Dex file recovered from the corresponding packed app (⊕, ⊖ and ⊙ represent the recovered Dex file that 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	⊙	⊕	⊕	⊕	⊕	⊙
DB-16	⊙	⊙	⊕	⊙	⊕⊖	⊕
DB-18	⊖	*Na*	⊙	⊕	⊕⊖	⊖

TABLE 5: Comparison among Android-unpacker, Android_unpacker, DexHunter and `PackerGrind` (`S` and `F` mean unpacking successfully and unsuccessfully, respectively).

Packer	Qihoo	Ali	Bangcle	Tencent	Baidu	Ijiami
Android-unpacker [15]	F	F	F	F	S	F
Android_unpacker [16]	S*	F	S	F	S*	F
DexHunter [78]	F	S	S	S	F	F
PackerGrind	S*	S	S	S	S*	S

* The packers Baidu and Qihoo reimplement specific methods in the packed apps using native code. Accordingly, the original Dalvik bytecode of these methods are removed and never released (details in Section 5.3.2). As a result, the unpackers cannot recover the bytecode of these methods from the packed apps.

After we replace these optimized Dex bytecode with the corresponding Dex bytecode, all the examples are disassembled by `Dex2Jar` successfully.

Second, we compare the difference between the original Dex files and the recovered Dex files of 170 randomly selected samples. First, for each sample, we implement a Python script, which first disassembles both the recovered Dex file and the original file, and then parses the classes, methods, and the Dalvik instructions from them. Afterwards, the differences between the classes, methods, and instructions in the original file and the corresponding recovered Dex file are compared. If differences are found, we further decompile both Dex files into Java code by leveraging the Dex to Java decompiler jadx [11] and then check the details of these differences through comparing the decompiled Java code manually. The details of the comparison results are summarized in Table 4.

The Dex files recovered from the apps packed by Qihoo and Ijiami in DB-15 as well as Ali and Tencent in DB-16 are the same as their original ones. Whereas, Ali adds two classes to the Dex files of the packed apps in DB-15 and each class has one field and three empty methods. Also, at the beginning of each Java method, an invocation of "*Exit.b(Exit.a())*" is inserted. Note that, *false* is always returned in *Exit.a()* and the function *Exit.b()* is empty. Moreover, in the apps packed by Tencent (i.e., DB-15), two addresses are added, and the apps packed by Ijiami (i.e., DB-16) have five additional classes.

For the apps packed by Bangcle, there are six and twelve additional classes inserted in DB-15 and DB-16, respectively. The Bangcle packer inserts a new method named *com_sec_plugin_action_APP_STARTED()* at the beginning of the *onCreate()* methods, which is first invoked when the packed apps start. However there are differences between the apps in DB-15 and DB-16. Specifically, in DB-15, an intent named *com.secneo.plugin.action.APP_STARTED* is created and broadcast in this new method, whereas a new monitoring thread is created in the packed apps of DB-16.

Except for inserting additional class to the packed apps, Baidu also re-implements the *onCreate()* methods using native code in the apps of DB-16 and DB-18 by default. Hence, although Baidu adds additional code to the packed apps, the original bytecode of the *onCreate()* methods in the apps of DB-16 and DB-18 are replaced with native code and cannot be recovered. Qihoo also protects the bytecode of specific methods through re-implementing the using native code in the packeds apps of DB-18. Since `PackerGrind` does not focus on converting the native code to bytecode, it cannot recover original bytecode of such protected methods. However, `PackerGrind` tracks the invocations of the framework APIs and thus the information about the invoked APIs can also further facilitate the static analysis tools.

In the packed apps of DB-18, Ijiami only releases the class related Dex data into memory just before the corresponding classes are defined to protect the Dex data in memory. Since `PackerGrind` collects the Dex data when they are used, it can collect the valid class related Dex data of all defined classes.

Answer to RQ2: `PackerGrind` can correctly recognize the packers used for protecting apps, and recover all Dex code that has not been 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 semantic information.

6.4 Comparison

We compare `PackerGrind` with three open-sourced unpacking tools, Android-unpacker [15], Android_unpacker [16], and DexHunter [78], which are from various research and industry communities. In this experiment, the correctness of the Dex files recovered from the apps packed by various packers are validated by a two-step checking way. First, we leverage Baksmali [6] to disassemble both the Dex files recovered from the packed apps and the Dex files from the original apps into smali code. Second, we compare the results of the recovered Dex files and that of the corresponding original Dex files. The failure in any step will lead to `F` in Table 5 indicating unsuccessfully unpacking.

As the unpacking results shown in Table 5, Android-unpacker leverages the debugging technique (e.g., *ptrace*) to inject into the process of the target packed app and then dump the Dex file in memory. Thus, it cannot be applied to the packed apps with anti-debugging capacity. Also, Android-unpacker determines the memory address of the target Dex file according to the signatures (i.e., specific string) of the packers. Specifically, Android-unpacker v1.2 supports four types of packers, including Bangcle, APKProtect, LIAPP, and Qihoo. For example, it determines Qihoo packer according to string "/libprotectClass" in the memory. However, since Android-unpacker cannot find the packers' signatures in the packed apps of DB-15, DB-16 and DB-18, it fails to unpack all the packed apps. Moreover, the latest version [2] of Android-unpacker is extended to support the packer Jaigu [15], but it still fails to unpack the packed apps.

2. Commit ID is 968ed234217edd16212738473f6493515e3d585b

Android_unpacker [3] is implemented through modifying Android 6.0. More precisely, it inserts unpacking code in the runtime methods *DexFile::DexFile()* and *OpenAndReadMagic()* in `dex_file.cc` to dump the Dex files to storage when the released Dex files are resolved by the ART runtime. Also, this unpacker focuses on dumping the Dex files that are dynamically released to the directory */data/data* and then loaded by the runtime. Since Ali packer does not release any Dex file to this directory, Android_unpacker unpacks the apps packed by Ali unsuccessfully. Because Tencent packer dynamically releases the protected Dex data and only an empty Dex file is loaded by the runtime when Android_unpacker dumps the Dex file from memory, it just outputs an empty Dex file from the apps packed by Tencent. Moreover, the Ijiami packer does not use these two methods (i.e., *DexFile::DexFile()* and *OpenAndReadMagic()*) to resolve the dynamically released Dex file and so that Android_unpacker cannot unpack the apps packed by Ijiami successfully. It is also worth noting that, since the packers Ali, Baidu, Ijiami, and Qihoo of DB-15 do not support Android 6.0, the apps packed by these packers in DB-15 cannot run on Android_unpacker.

DexHunter [4] fails to unpack the apps packed by Qihoo, Baidu and Ijiami because it just inserts unpacking code to the special runtime functions through directly modifying the Android runtime. More precisely, since Qihoo leverages its customized code to load the class information in the Dex files instead of invoking runtime functions, the Dex files recovered by DexHunter contain no valid Dex data in the original Dex files. Moreover, Baidu fills the Dex header of the Dex files in the packed apps with invalid data and hence Dex files recovered from its packed samples cannot be disassembled by the exiting tool correctly. DexHunter does not adapt to the apps packed by Ijiami because Ijiami detects analysis and unpacking tools through time checking. Particularly, Ijiami checks the time consumed by the customized special task. If the consumed time exceeds the threshold, it regards it as being analyzed and exits. Since DexHunter needs to stop and search for the target content in the memory, which is a time-consuming task, it can be detected by the packer and does not adapt to the apps packed by Ijiami.

From the unpacking result, we can find the apps packed by all six packers are successfully unpacked by `PackerGrind`. Note that, specific methods in the apps packed by Qihoo and Baidu are re-implemented with native code (details in Section 5.3.2). Although `PackerGrind` does not focus on analyzing such native code, it tracks the framework APIs invoked by the native code, which can also facilitate further static analysis.

Answer to RQ3: `PackerGrind` can be applied to more sophisticated packers and achieve better unpacking results than other available unpacking tools (i.e., Android-unpacker, Android_unpacker and DexHunter).

6.5 Unpacking Malware

We apply `PackerGrind` to 399 packed malware samples including 389 samples packed by twelve popular known packers and 10 samples packed by unknown/other packers. Moreover, there are 200 samples collected before 2017 and 199 samples found in 2017 and 2018. `PackerGrind` can successfully recover the Dex files from all of them. After vetting both the recovered and original Dex files , we find the packers are usually leveraged to hide the payload of the malware, especially the invoked sensitive framework APIs as well as the required permissions. Consequently, to evaluate the assistance introduced to the existing static analysis tools by `PackerGrind`, we compared the sensitive APIs as well as permissions in recovered Dex files and original Dex files. Specifically, given a Dex file, we first search all sensitive APIs in it and then count the numbers of the invoked sensitive APIs and the permissions following the mechanism of PScout [22]. Note that various static analysis tools are already proposed to analyze and detect malware based on sensitive APIs and permissions [23], [62], [70], [76], [77].

In this experiment, we leverage P_p/A_p and P_r/A_r to represent the number of the detected sensitive permissions/APIs in the Dex files of the packed apps and the recovered Dex files, respectively. For the apps packed by twelve various packers, the average values of P_p, A_p, P_r, and A_r are shown in Table 6, which intuitively shows the recovered Dex files expose more sensitive permissions and APIs than the Dex files in the packed apps. Consequently, after being unpacked by `PackerGrind`, more sensitive behaviors are exposed to the static malware analysis and detection tools. Take the apps packed by Ali as examples, there are 44.76 sensitive APIs exposed in the recovered Dex files but no sensitive APIs are detected in the Dex files of the packed apps.

Answer to RQ4: `PackerGrind` can efficiently expose the hidden behaviors of the packed malware through unpacking and the unpacking results effectively facilitate the existing malware detection and analysis mechanisms.

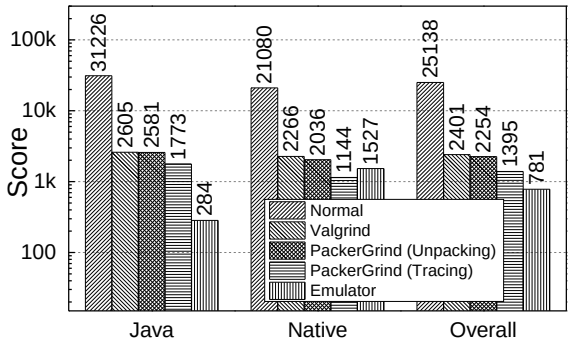

Fig. 8: CF-Bench results (high score means high performance).

6.6 Overhead

To evaluate the overhead introduced by `PackerGrind`, we run CF-Bench [7] on Nexus 5 without instrumentation, with only Valgrind, with dynamic tracing, and with unpacking as well as on the Android emulator Qemu, respectively. We first leverage the scores achieved by CF-Benchmark without the Valgrind framework running on the smartphone as the baseline. Then we run CF-Benchmark on the smartphone with only Valgrind framework and `PackerGrind` (including both

3. Commit ID is 971b30e005240020383891c52148463801e1f6d8
4. Commit ID is 9d829a9f6f608ebad26923f29a294ae9c68d0441

TABLE 6: 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	Netease	Others
Number of samples	29	24	109	34	68	12	62	9	2	18	11	11	10
Average value of P_p	0.00	2.65	1.32	0.24	0.03	0.00	0.94	7.22	0.00	4.28	0.82	8.18	2.60
Average value of P_r	4.83	3.65	8.34	7.56	3.41	3.00	10.08	7.78	0.50	10.33	10.36	17.36	3.90
Average value of A_p	0.00	5.62	1.45	0.88	0.04	0.00	2.81	40.67	0.00	19.56	2.64	12.27	4.70
Average value of A_r	44.76	9.38	34.46	50.18	18.24	14.33	54.76	58.00	4.50	90.94	33.45	43.27	7.00

tracking and unpacking) to evaluate the additional overhead incurred by them. Since there are multiple dynamic analysis tools or unpackers that are implemented on the Android emulator, such as DroidUnpacker [31], DroidScope [73], NDroid [72], and Android_unpacker [16], which are all implemented based on the Android emulator Qemu but for different versions of Android systems. To avoid the bias introduced by the differences of Android systems, we evaluate the performance of the Android emulator Qemu running Android 6.0, which is the same as the system running in Nexus 5 smartphone. Meanwhile, the host runs Ubuntu 16.04 system and is equipped with Intel(R) Xeon(R) CPU E5-2620 and 32 GB memory.

As shown in Fig. 8, three types of scores (i.e., overall, Java, and native) are compared under five various configurations. For the overall scores, PackerGrind introduces 11 and 18 times slowdown to the target app during tracing and unpacking, respectively, and there is already 10 times slowdown incurred when only Valgrind framework runs without any tracing and unpacking actions. Hence, compared with the DBI framework Valgrind, the tracing and unpacking tasks of PackerGrind just incur 1.07 and 1.72 more times slowdown to the analyzed apps. Compared with the Nexus 5 smartphone, the emulator incurs more than 32 times overall slowdown to the target apps. Note that, emulator-based analysis tool also incurs much additional slowdown to the emulator when they are leveraged to carry out the dynamic analysis or unpacking tasks [31], [73]. In addition, for the sample of new Android packer, PackerGrind only needs to analyzing it once, and then all apps packed by the same packer can be quickly unpacked by PackerGrind according to the identified data collection points.

Answer to RQ5: PackerGrind introduces reasonable additional overhead during either tracking or unpacking.

7 Discussion

PackerGrind can only recover the Dex data after they are dynamically loaded into the memory. Although the majority of existing packers dynamically release all protected Dex Data at the beginning, there are also packers that release the bytecode of a method just before the method is invoked. Currently, we have adopted IntelliDroid [68] to trigger the execution of such methods and we will leverage more advanced input generators for Android [29], [46] to enhance PackerGrind in future work.

Since PackerGrind is based on Valgrind, which also have hinted in memory, such as starting command, and thus the packers may detect the presence of PackerGrind and then stop releasing the real code. However, to protect the DBI framework Valgrind from being detected, we can hook the system library functions related to string matching (e.g., *strcmp()*) to change their results. Also, we can force the app to release the code through inserting IR statements to change the conditions of the conditional branches.

The advanced packers may adopt virtual-machine-based (i.e., VM-based) techniques to translate the Dalvik bytecode into another type of instructions and provide a customized VM to execute them. PackerGrind currently cannot be applied to recover the Dalvik bytecode protected by VM-based techniques because it mainly focuses on recovering the Dalvik bytecode that are released in memory. However, PackerGrind can still track the Java methods invoked by the VM through *JNI* reflection. In future work, we will enable PackerGrind to recover more semantics of the protected Dalvik bytecode according to the behaviors of VM.

8 Related Work

In this section, we present the related work on unpacking native programs and Android apps.

8.1 Unpacking Native Programs

Many packing/unpacking related studies have been conducted but they usually focus on the native programs [27], [36], [57], [66]. Although both the native unpackers and Android unpackers unpack the target programs based on dynamic tracking mechanisms, the native unpacking techniques cannot be applied to unpacking Android apps *directly* because both their unpacking targets and unpacking results are different. More precisely, since native unpackers aim to dump the protected native instructions, they focus on tracking the executions of instructions and monitoring memory modifications in the native layer. For example, the unpacker presented in [66] leverages TEMU [59] , a dynamic analysis tool based on the emulator Qemu [26], to monitor the execution of the target processes. Renovo [41] also monitors the instructions and tracks memory operations depending TEMU. Saffron [55] builds a unpacking tool based on Intel's PIN and it unpacks native programs through dynamically monitoring the control transfers or modifications of memory.

In contrast, the Android unpackers aim to unpack the protected Dalvik bytecode and other Dex items, such as `class_data_item`, and then assemble them into valid Dex files as the outputs. Since these protected bytecode are released by the packer in native layer and then parsed and executed by the Android runtime, PackerGrind needs to track

the packing behaviors in multiple layers. Moreover, besides tracking the behaviors related to the bytecode, PackerGrind also needs to monitor the behaviors related to other Dex items, which will be used to assemble the valid Dex files.

Many packing/unpacking related studies have been conducted but they usually focus on the x86 architectures [27], [36], [57], [66], and thus these techniques do not adapt to the packed Android apps [75], [78]. Except the difference between the formats of the Android apps and the x86 desktop programs, they also have different architectures and execution models [75], [78]. For example, the Android packers should consider both the bytecode and the native code if any in the apps, and but the traditional packers just need to take into account the native code [27], [36], [57], [66].

8.2 Unpacking Android Apps

Since the malicious Android developers start to adopt packers to hide the malicious content of malware, a few studies have been carried out on the Android packers by both academia [31], [43], [50], [69], [75], [78] and industry [4], [15], [16]. Specifically, DWroidDump collects Dex data when *dvmDexFileOpenFromFd()* is invoked because this function is used to map the Dex file to memory [43]. DexHunter inserts code in *defineClassNative()* to extract Dex files from memory [78]. Similarly, the Android_unpacker is implemented through inserting Dex data dumping code in runtime functions *DexFile:DexFile()* and *OpenAndReadMagic()* [16], which are used to resolve the Dex file in memory or from storage, respectively. Moreover, DexLego modifies the Dalvik interpreter to dump the interpreted Dalvik instructions into storage [50] depending on the system library functions, such as *open()* and *write()* of `libc.so`. Tiro [69], CrackDex [39], and DexX [61] are also implemented through modifying the Android runtime (i.e., DVM or ART) and then dump the Dex data when the Dex files are resolved. Android-unpacker needs to inject itself into the target process leveraging debugging mechanism (i.e., ptrace) and then determines the memory region of the Dex data through looking up specific strings [15]. Also, the GDB built for ARM is leveraged to unpack the apps packed by DexProtect and Bangcle in [45]. DumpAPK leverages Xposed [13] to hook runtime functions *openDex()* and *defineClass()* to collect the Dex data when the Dex files are resolved using these two functions [4]. In addition, except unpacking packed apps, DroidUnpack can also track the behaviors of Android packers by leveraging the virtual machine instrumentation techniques and it is implemented based on the Android emulator Qemu [31].

However, the existing unpackers have various limitations when they are applied to unpacking the apps packed by modern packers in practice. First, since most of them adopt the one-pass strategy to dump the Dex data at fixed predefined points, such as DexHunter, Android_unpacker and DumpAPK, the packers can evolve the packing behaviors and then evade these unpacking tools straightforwardly. Second, the unpackers (e.g., Dexhunter, Android_unpacker, DexLego) that need to modify Android systems and dump the Dex data in memory to storage depending on the system library functions (e.g., *open()* and *write()*). However, there are multiple packers that hook these functions to prevent the Dex data from being dumped, and thus these unpackers cannot handle the apps packed by the packers with the anti-dump capacity. For example, since Bangcle hooks the system library functions (e.g., *open()*, *write()*, and *close()*) to protect the Dex data from being dumped, these unpackers cannot be applied to the apps packed by Bangcle. Third, some sophisticated packers (e.g., Ijiami) check whether the packed apps run in an emulator or debugging environment before releasing the protected Dex data. If so, the packers exit immediately. As a result, the debugger or emulator-based unpackers (e.g., DumpAPK and DroidUnpack) cannot process the apps packed by these packers with anti-emulator or anti-debug capacities.

PackerGrind leverages an adaptive approach to unpack the packed apps through tracking the packing behaviors in multiple layers and thus it can be applied to the apps packed by new and evolved packers. In addition, it runs on real devices and leverages the methods provided by Valgrind to dump the memory data. Hence, although the packers with the capacity of anti memory dumping, debugging and emulator, PackerGrind can still handle their packed apps. Also, PackerGrind supports both DVM and ART runtime, as well as the evaluation results show it outperforms the other unpackers.

Dynamic Tracking Tools: Existing cross-layer monitoring tools [32], [53], [67], [73] for Android can neither collect all necessary information nor fulfill the requirement for handling packed apps because of various limitations. Such as ProfileDroid [67] that relies on apktool to conduct static analysis, and thus it does not adapt to the packed apps. TaintDroid [32] just works in the runtime layer of DVM and does not support ART. Since both DroidScope [73] and NDroid [72] are implemented based on the Android emulator (i.e., Qemu) and hence they can be detected by packers [40].

This paper is an extension of the conference paper [71] with many improvements to our unpacking system. First, the packers are becoming more sophisticated and ART has already been the default runtime from Android 5.0 (released in June 2014) and shared more than 90% Android market by 2019. Hence, we improve the capacity of PackerGrind for ART runtime, such as collecting Dex data and tracking packing behaviors, which involve loading Oat files, parsing Dex data, executing methods, loading native code loading, and registering native code in Section 5.2.1. Second, we extend PackerGrind with the capacity to generate the signatures for packers by exploiting their unique packing/protection information. Consequently, given a target app, PackerGrind first checks whether it is hardened by the packers whose samples have been analyzed before. If so, PackerGrind unpacks the app using the recorded data collection points for this packer, thus speeding up the unpacking process. Third, in [71], PackerGrind collects the Dex data once but advanced packers may release different parts of a method' bytecode at different call sites. We enhance PackerGrind to tackle this issue by tracking all invoked Java methods so that it can collect all valid bytecode of the Java methods when they are invoked. Last but not least, we use much more samples packed by the evolving commercial Android packers and the newly collected packed malware for evaluating PackerGrind in Section 6

9 Conclusion

After studying the mechanisms leveraged by the existing Android unpacking tools, we find they adopt the one-pass strategy to unpack Android apps but the Android packers evolve quickly, and thus such unpacking tools cannot be applied to the apps packed by the evolved and new packers. To address this issue, we propose a novel adaptive way to unpack the Android apps according to the packing behaviors. Specifically, we adopt an iterative way to track the packing behaviors used by the packers and obtain the hidden Dex data according to the concrete packing behaviors instead of dumping Dex data at fixed points. Hence our unpacking approach also adapts to the evolved and new packers. We also implemented the unpacking tool named `PackerGrind` based on this unpacking approach and evaluated its effectiveness and efficiency with both open sourced Android apps and real malware.

References

[1] "Apk protect," https://sourceforge.net/projects/apkprotect.
[2] "Cisco 2014 annual security report," https://goo.gl/7524ER, 2014.
[3] "OWASP mobile top 10 risks," https://goo.gl/DC8nvN, 2014.
[4] "Dumpapk," https://github.com/CvvT/DumpApk, 2015.
[5] "F-Droid," https://f-droid.org/, 2015.
[6] "Baksmali," https://github.com/JesusFreke/smali, 2016.
[7] "CF-Bench," http://http://bench.chainfire.eu/, 2016.
[8] "Dex2jar," https://github.com/pxb1988/dex2jar, 2016.
[9] "Dexdump," https://goo.gl/eDpDzi, 2016.
[10] "IDA Pro," https://www.hex-rays.com/products/ida/, 2016.
[11] "Jadx," https://github.com/skylot/jadx, 2016.
[12] "monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/index.html, 2016.
[13] "Xposed module repository," http://repo.xposed.info, 2016.
[14] "ZjDroid," https://github.com/halfkiss/ZjDroid, 2016.
[15] "Android-unpacker," https://github.com/strazzere/android-unpacker, 2019.
[16] "Android_unpacker," https://github.com/CheckPointSW/android_unpacker, 2019.
[17] "Statistics and facts about Android," https://goo.gl/eLubgk, 2019.
[18] S. Aimoto, "Five ways Android malware is becoming more resilient," https://goo.gl/cdN2Vi, 2016.
[19] Alibaba Inc., http://jaq.alibaba.com/.
[20] Android, "Gestureoverlayview," https://developer.android.com/reference/android/gesture/GestureOverlayView.html, 2019.
[21] A. Apvrille and R. Nigam, "Obfuscation in Android malware, and how to fight back," *Virus Bulletin*, July 2014.
[22] K. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM CCS*, 2012.
[23] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. ACM/IEEE ICSE*, 2015.
[24] Baidu Inc., http://app.baidu.com.
[25] Bangcle Inc., http://www.bangcle.com/.
[26] F. Bellard *et al.*, "Qemu open source processor emulator," http://www.qemu.org, 2017.
[27] 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 *Proc. ACM CCS*, 2015.
[28] 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 *Proc. USENIX Security*, 2015.
[29] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proc. IEEE/ACM ASE*, 2015.
[30] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.
[31] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation." in *NDSS*, 2018.
[32] 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.
[33] 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 *Proc. ISSRE*, 2016.
[34] Gartner, Inc., "Debunking six myths of app wrapping," https://goo.gl/zEzJn6, 2015.
[35] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *Proc. NDSS*, 2015.
[36] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proc. RAID*, 2008.
[37] Ijiami Inc., http://www.ijiami.cn/.
[38] N. Inc., https://dun.163.com/product/android-reinforce.
[39] Z. Jiang, A. Zhou, L. Liu, P. Jia, L. Liu, and Z. Zuo, "Crackdex: universal and automatic dex extraction method," in *Proc. IEEE ICEIEC*, 2017.
[40] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect Android emulators," in *Proc. ACSAC*, 2014.
[41] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. ACM WORM*, 2007.
[42] G. Kelly, "97% of mobile malware is on Android. this is the easy way you stay safe," https://goo.gl/z121Sq, 2014.
[43] D. Kim, J. Kwak, and J. Ryou, "Dwroiddump: Executable code extraction from android applications for malware analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 9, 2015.
[44] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. ACM/IEEE ICSE*, 2015.
[45] J. Lim and J. H. Yi, "Structural analysis of packing schemes for extracting hidden codes in mobile malware," *EURASIP Journal on Wireless Communications and Networking*, vol. 2016, no. 1, 2016.
[46] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of Android applications," in *Proc. ACM/IEEE ICSE*, 2016.
[47] NAGA IN Inc., http://www.nagain.com/.
[48] N. Nethercote and J. Seward, "Valgrind: a framework for heavy-weight dynamic binary instrumentation," in *Proc. ACM PLDI*, 2007.
[49] Netqin Inc., https://www.netqin.com.
[50] Z. Ning and F. Zhang, "Dexlego: Reassembleable bytecode extraction for aiding static analysis," in *Proc. DSN*, 2018.
[51] Palo Alto Networks, "Wildfire cloud-based threat analysis service," https://goo.gl/2SJSRi, 2016.
[52] PayEgis Inc., http://www.payegis.com/.
[53] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proc. IEEE DSN*, 2014.
[54] Qihoo360 Inc., http://dev.360.cn/.
[55] D. Quist, "Valsmith. covert debugging: Circumventing software armoring," in *Proc. of BlackHat*, 2007.
[56] P. Sabanal, "Hiding behind art," *Blackhat*, 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*, 2016.
[58] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.
[59] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.
[60] T. Strazzere and J. Sawyer, "Android hacker protection level 0," *DEFCON*, 2014.
[61] C. Sun, H. Zhang, S. Qin, N. He, J. Qin, and H. Pan, "Dexx: a double layer unpacking framework for android," *IEEE Access*, vol. 6, 2018.

[62] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, 2017.
[63] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors." in *Proc. NDSS*, 2015.
[64] Tencent Inc., https://www.qcloud.com/product/cr.
[65] TIS, "Tool interface standard (tis) executable and linking format (elf) specification version 1.2," *TIS Committee*, 1995.
[66] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE S&P*, 2015.
[67] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *Proc. Mobicom*, 2012.
[68] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. NDSS*, 2016.
[69] ——, "Tackling runtime-based obfuscation in android with {TIRO}," in *Proc. USENIX Security*, 2018.
[70] 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*, vol. 49, no. 2, 2016.
[71] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. IEEE/ACM ICSE*, 2017.
[72] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2019.
[73] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Security*, 2012.
[74] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in Android apps," in *Proc. SecureComm*, 2015.
[75] 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 *Proc. RAID*, 2015.
[76] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proc. IEEE DSN*, 2016.
[77] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. ACM CCS*, 2014.
[78] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed Android applications," in *Proc. ESORICS*, 2015.
[79] M. Zheng, M. Sun, and J. C. Lui, "DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability," in *Proc. WCMC*, 2014.
[80] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE S&P*, 2012.
[81] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 867–876.