Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation

Yue Duan*, Mu Zhang†, Abhishek Vasisht Bhaskar‡, Heng Yin*,
Xiaorui Pan§, Tongxin Li¶, Xueqiang Wang§, and XiaoFeng Wang§

*University of California, Riverside †Cornell University
‡Grammatech. Inc. §Indiana University Bloomington ¶Peking University

yduan005@ucr.edu, mz496@cornell.edu, abhaskar@grammatech.com,
heng@cs.ucr.edu, {xiaopan, xw48, xw7}@indiana.edu, litongxin@pku.edu.cn

***Abstract*—The prevalent usage of runtime packers has complicated Android malware analysis, as both legitimate and malicious apps are leveraging packing mechanisms to protect themselves against reverse engineer. Although recent efforts have been made to analyze particular packing techniques, little has been done to study the unique characteristics of Android packers. In this paper, we report the first systematic study on mainstream Android packers, in an attempt to understand their security implications. For this purpose, we developed DROIDUNPACK, a whole-system emulation based Android packing analysis framework, which compared with existing tools, relies on intrinsic characteristics of Android runtime (rather than heuristics), and further enables virtual machine inspection to precisely recover hidden code and reveal packing behaviors. Running our tool on 6 major commercial packers, 93,910 Android malware samples and 3 existing state-of-the-art unpackers, we found that not only are commercial packing services abused to encrypt malicious or plagiarized contents, they themselves also introduce security-critical vulnerabilities to the apps being packed. Our study further reveals the prevalence and rapid evolution of custom packers used by malware authors, which cannot be defended against using existing techniques, due to their design weaknesses.**

I. INTRODUCTION

Mobile computing has become a new frontier for the perpetual battle between cybercriminals and those who want to stop them. For years, those criminals are utilizing all kinds of malicious apps to gain undesired access to system resources [19], [25], [23], collect private user information [22], [21], [26], [44], [46], compromise data integrity [45], [28], etc. In response, various static [15], [37] and dynamic analysis techniques [21], [39] have been developed and deployed to capture their malicious activities. Such protection, however, has come under the threat of Android app packing, which becomes increasingly popular. Studies [43], [40] show that both malicious and benign apps utilize packing techniques to hide their code. The complexity in analyzing obfuscated code, as introduced by these techniques, has become a new barrier to protecting Android users. Particularly, without in-depth understanding of these Android packers, malicious, vulnerable and plagiarized apps could easily circumvent the vetting process put in place by app markets and spread across Android devices through these markets.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23296
www.ndss-symposium.org

Understanding packers. Despite the importance of this emerging trend (app packing), no comprehensive study, however, has ever been conducted to help the community understand the status quo of Android packing and unpacking techniques, which is crucial to building practical defense and mitigating the security risks brought in by these techniques. In this paper, we report our study on the problem, the first of this kind up to our knowledge. The study investigates a broad spectrum of Android packers and characterizes the apps utilizing them in terms of their security implications. More specifically, we seek answers to a set of security-critical questions, which *has never been addressed by the prior research*, as follows.

First of all, we want to find out how today's Android packers are being used, particularly by cybercriminals. *Are they (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware? What are the distributions of different commercial and custom packers across Android apps? How do the distributions change over time?*

Then, we look into technical details. *How do Android packers work? Is it very different from traditional packing? What are the security impacts when applying the packers to apps? Is it easy for malicious developers to exploit commercial services to pack their malware or plagiarized apps?*

Moving forward, we study the direction of technique development and its security implications. *Have Android packers*

been evolving and how? What are the future trends of the techniques?

Finally, we check the state-of-the-art of Android unpacking techniques. Particularly, *How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?*

Answers to these questions can only be found through an in-depth analysis of packing and unpacking operations on Android code, to reliably identify related behaviors including those never seen before. This cannot be done by any existing Android unpackers [43], [40], [33], which can only handle *known* packing operations and have no view for behaviors at native level. Although the tools built for unpacking PC programs (e.g., Renovo [27]) could help find some new packers, they are just designed for binary code and cannot handle Java code. So far, none of the existing techniques are capable of performing the cross Java and native code analysis required for an in-depth understanding of complicated Android packing behaviors.

Our study and findings. To find answers to these security-critical questions and better understand the security implications of Android packing techniques, we developed an Android packing analysis framework called DROIDUNPACK based on a whole-system emulation. To reliably capture and analyze unpacking behaviors on Android, DROIDUNPACK has been designed to monitor at the lowest level and reconstruct Java-level execution. In this way, it can catch the intrinsic "write-and-then-execute" unpacking behaviors at either native level or Java level or both.

With the help of this analysis framework, we conducted a comprehensive study over 6 major commercial packers, 3 state-of-the-art unpackers and 93,910 Android malware samples in the wild. Here we highlight some interesting discoveries made in our research:

(1) Android packers have been heavily abused by malware developers. When commercial packers are gaining popularity, the ratio of the malware samples that are packed by these packers also goes up from 2010 to 2015.
(2) Android commercial packers can be easily leveraged to pack malware and plagiarized apps, making detection of such apps much harder.
(3) Some packers introduce severe security vulnerabilities to the apps they pack, which can lead to data breaches and arbitrary code execution. These serious problems were found to affect more than 1 billion users.
(4) Android packers are quickly evolving with new behaviors in the past few years, which renders even state-of-the-art unpackers less effective.

Contributions. The contributions of the paper are summarized as follows:

- We designed and implemented a novel tool called DROIDUNPACK that automatically reconstructs semantic views from multiple levels of the Android system and reliably captures packing behaviors through four different analyzers. We plan to make this new tool publicly available for a continuous analysis on app packing techniques.
- We performed the first large-scale measurement study on all mainstream Android (un)packers and a massive number of apps over a large time frame (from 2010 to 2015). Our study has brought to light new understandings and insights about these (un)packing techniques and their ecosystem (e.g. new packing techniques, their evolution, etc.), which is invaluable for effectively mitigating the security risks they introduce.

Roadmap. The rest of the paper is organized as follows: Section II presents the uniquenesses about Android packing; Section III elaborates the design and implementation of our analysis platform DROIDUNPACK; Section IV describes the research methodology of our large-scale study; Section V elaborates our study and findings; Section VI surveys the related prior research, and Section VII concludes the paper.

II. UNIQUENESSES ABOUT ANDROID PACKING

In this section, we describe three major differences between Android and traditional PC that impact the study of packing. These uniquenesses about Android motivate our study and show the technical challenges in DROIDUNPACK.

A. System, Runtime & Apps

Unlike traditional PC, Android system has a multi-level design. It is built on top of a customized Linux kernel. A process named Zygote is the parent for all Android app processes. Above the kernel, Android system provides a set of libraries including app runtime. The runtime coordinates apps with Android framework libraries so that the apps can interact with lower-level system through framework APIs. This fundamental design difference in Android system requires our tool to have multi-level views about the whole system including OS level, binary level and Java level views.

Moreover, just like Android system, Android runtime environment is also very different from traditional PC, and has changed drastically over time.

Dalvik virtual machine. Legacy Android (version 4.4 and earlier) leverages Dalvik Virtual Machine (DVM) to interpret DEX bytecode programs at runtime. At install time, a `dexopt` tool optimizes the input DEX bytecode and creates `ODEX` files so as to improve runtime efficiency. Upon execution, DVM enables bytecode interpretation and translates DEX code to native code for the target architecture.

ART environment. The recent Android system (version 5.0 and later) has substituted Dalvik VM with the new Android Runtime (ART) in order to improve runtime performance. In contrast to the bytecode interpretation in DVM, ART conducts ahead-of-time (AOT) compilation to produce machine dependent code prior to execution. To do so, ART utilizes the compiler, `dex2oat`, to transform an input DEX executable into an OAT file. Internally, `dex2oat` can perform multiple rounds of optimizations and depending upon the existence of legacy code, it may select between "interpret" and "quick" modes to achieve different levels of optimizations. The "interpret" mode means no code will be compiled into native, while "quick" mode compiles as many codes as possible.

As a result, Android apps are designed to be quite different from traditional PC programs as well. Android apps are built

as a combination of distinct components that can be invoked individually and can contain both Java and native parts. Native components are simply shared libraries that are dynamically loaded at runtime. The app runtime library (`libdvm.so` or `libart.so`) interprets or compiles the Java components to produce and launch native instructions. The Java Native Interface (JNI) is then used to enable communications between the native and Java sides. Thus, a packed Android app often packs its Java code as well as its native code (major program logic or critical functionality) into binary resource files. Usually, it still maintains a dummy Java component, which acts solely as a dispatcher to launch the unpacking procedure.

B. Unpacking techniques

Runtime packers in general have been well studied and series of solutions have been proposed to defeat them [30], [29], [27], [31]. However, due to the differences in so many aspects, there exists a major discrepancy in the unpacking techniques between Android and PC.

PC unpackers such as Omniunpack [29] and renovo [27] monitor and trace the packed program execution at native instruction or page granularity using either memory protection mechanism or emulated environment so that they can reliably uncover the program behaviors at native level. Nonetheless, this kind of unpacking technique does not fit into Android scenario where apps contain both native and Java components. Fundamentally, the design lacks the capability of monitoring Java level behaviors, thus, will not be able to understand anything happens at that level.

On the other hand, existing Android unpackers [43], [40], [33] fall short of native side. Current Android unpackers can be roughly categorized into three types to extract code based on different design choices: 1) signature-based memory dump unpacker as Kisskiss [33]; 2) hooking-based memory dump unpacker as DexHunter [43]; 3) Dalvik data structures dumping and DEX file assembly unpacker as AppSpear [40]. All the Android unpackers rely on Java level information other than intrinsic nature of packed programs, which is, the original code will be dynamically generated and then executed [27]. As a result, none of them is able to detect and analyze *previously unknown* packing techniques and understand what happens at the native side let alone the interactions between Java and native.

C. Android Ecosystem

Last but not least, another significant and special characteristic about Android which has great influence on packing study is the unique Android ecosystem. This ecosystem applies a huge impact to both app developers and users. After the Android apps are developed, the developers upload them to Android app markets, e.g., Google Play. The market will perform necessary vetting process to the uploaded apps and make them available for the end users.

By default, Android system disallows users to install apps outside of Google Play. Further, because of the vetting performed by those Android app markets, users tend to trust them and download apps from them. According to 2016 Google I/O [3], Google Play has reached over 1 billion monthly active users in 2016 which makes it the world largest app distribution platform. As a result, malware and plagiarized apps that circumvent app markets security checks and infiltrate into this ecosystem can impose even bigger threat to users than normal malware. However, according to prior reports [4], [8], packing techniques can indeed help malicious developers sneak malware into Google Play. This fact motivates us to study Android packing techniques.

What's more, unlike traditional PC, commercial packing services have become a part of the Android ecosystem as well. They are being widely used by many developers to pack and protect their intellectual property before submitting to app markets [40]. In order to prevent people from abusing the services, they have enforced their own malware and plagiarism detection mechanism. Our study also would like to find out if these services can be exploited and abused by malicious users. And since all the Android commercial packing services are freely available to users, it gets us wondering about what their business models are and further motivates us to study the detailed behaviors of those packers.

III. DroidUnpack System

A. Key Idea

To address the unique challenges in detecting and analyzing unpacking behaviors in Android, we need to:

(1) Monitor app execution at the lowest level, so we do not miss any behaviors related to unpacking;
(2) Reconstruct Java level execution, for accurate detection and better understanding of unpacking behaviors.

To capture the intrinsic characteristics (i.e, Write-and-then-Execute) of unpacking, we will monitor the app execution at the native code level to label dirty memory regions, as well as code execution happens at both native and Java levels. In this way, we are able to detect and analyze unpacking behaviors happening at either level or in a combination of both.

To do so, we take a whole-system emulation based approach. More specifically, we run the android system and the app of interest within an emulator so that we can easily monitor all memory writes initiated by the app. Then by reconstructing the Java execution context from native execution, we are able to reliably detect the execution of unpacked code, no matter if the unpacked code is interpreted, pre-compiled, or just native code.

B. DroidUnpack Overview

To realize this key idea, we choose to build DroidUnpack on top of DroidScope [39]. DroidScope is a QEMU and VMI-based dynamic instrumentation framework that enables instruction tracing on both Linux and DVM sides. However, it does not support the recent Android Runtime (ART), and thus cannot recover the high-level code semantics in ART. To address this limitation, we manage to reconstruct the ART view of running Android apps. Figure 1 illustrates the overview of DroidUnpack.

The entire Android system, including the packed Android apps, runs on top of an emulator, and the analysis and unpacking are completely conducted from outside of the emulator. We introspect the guest Android system, so that both the OS-level and ART-level semantics can be reconstructed using trustworthy points-to relations among internal data structures.

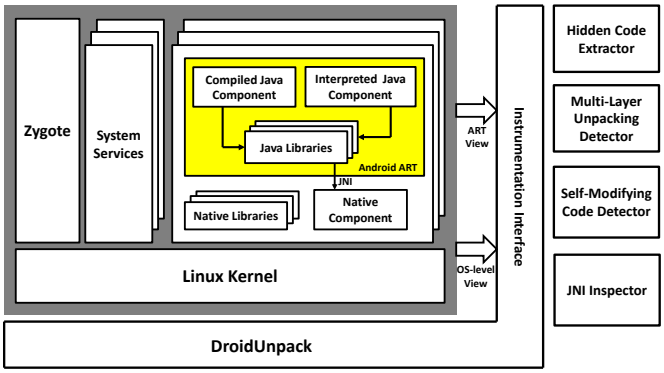

Fig. 1: Overview of DROIDUNPACK.

Interfacing with the core DROIDUNPACK platform, we have developed several analysis tools to investigate packed Android programs. 1) The *Hidden Code Extractor* precisely identifies and dumps memory regions that contain hidden DEX, OAT methods. 2) The *Multi-layer Unpacking Detector* discovers iterative unpacking operations that intermittently occur in multiple layers. 3) The *Self-Modifying Code Detector* detects an even stealthier unpacking behavior that intentionally wipes out previous executable code. 4) The *JNI Inspector* aims to search for sensitive API calls made through JNI interface.

C. Reconstructing Semantic View

Semantic view consists of two views at different levels, OS-level view and ART-level view. We rely on DroidScope to recover the OS-level view which provides two types of information: 1) the native process names and 2) the meta-data of memory-mapped modules for each process (i.e., base address, size, name, corresponding inodes and function offsets). Hence, we can accurately pinpoint a native function in memory via matching its address with $(module\_base\_address + offset)$.

We modify DroidScope to support the reconstruction of ART-level semantics. In particular, we have managed to recover the application names, compiled and interpreted Java methods.

Application name. The application name plays an important role in Android app unpacking because it indicates the context of decrypted hidden code. Unlike native processes, the name of an Android application is not resolved when a `fork` takes place. Instead, its name is later appointed through calling the native function `set_process_name`. Hence, we hook this function in the native library `libcutils.so` in order to correlate application name to each individual app process.

Compiled Java method. We further recover the corresponding Java method names, offsets and sizes of native functions that have been pre-compiled from DEX code. Such information can assist accurate collection and semantic-level understanding of the code.

To collect such meta-data for compiled methods, we need to search for the associated DEX and OAT files. To do so, we first hook the function `ArtMethod::Invoke()` in `libart.so`, which accesses `ArtMethod` code for invocation. Next, at the hooking point, we retrieve the `ArtMethod` data structure from memory, which contains a reference `HeapReference<Class> declaring_class_` that eventually points to its host class. Using the reference, we locate the class structure that holds the resolved DEX file cache `DexCache`. Then, we can reverse engineer this DEX cache to obtain the pointer to `DexFile` data structure.

Algorithm 1 Locating Executable in Memory

```
1:  procedure LocateCodeInMem(pc)
2:      mod ← GetCurrentModule(pc)
3:      if mod == "libart.so" then
4:          func ← GetCurrentFunction(pc)
5:          if func == "DoCall(ArtMethod*)" then
6:              md ← GetDexMeth(ArtMethod*)
7:          else if func == "ArtMethod::Invoke()" then
8:              md ← GetNativeMeth(ArtMethod*)
9:          end if
10:         mem_method ← GetAddressRange(md)
11:         return mem_method
12:     end if
13:     return ∅
14: end procedure
```

Once a `DexFile` has been discovered, DROIDUNPACK can further identify the code module that hosts this DEX file. This code module is in fact the OAT file that contains each original DEX file as well as its compiled `OatClass`es. By walking through each `OatClass`, we can then retrieve its meta-data, including the name and offset of every `OatMethod`. In this way, we reconstruct the mapping between name and address (i.e., $(module\_base\_address + offset)$) for each compiled method. Besides, we also iterate over every `OatMethodHeader` to find the code size of corresponding `OatMethod`, so that at runtime, we can precisely dump each unpacked code at method level.

Interpreted Java method. Although Android ART runtime provides the capability of compiling all Java methods beforehand, bytecode interpretation still remains available. Therefore, we also need to handle interpreted methods and retrieve their semantic information. Similarly, we hook the `DoCall()` function in `libart.so`, which starts the interpretation of `ArtMethod`s in Java. Again, we can trace back to the corresponding `DexFile` from each `ArtMethod`. In addition, we also obtain the `dex_method_index_` of `ArtMethod`, with which we can identify the exact `DexMethod` in the `DexFile` and therefore extract its name, offset, size and bytecode instructions. In this way, we are able to capture the interpreted Java code that is unpacked during execution.

D. Code Behavior Analysis

Powered by the unique capability of DROIDUNPACK, we have enabled four code analyzers to understand Android packer behavior.

Hidden OAT/DEX code extraction. Malicious code is packed to avoid detection and analysis. With the reconstructed OS view and ART-level view, we can now extract packed executable code at runtime. We first follow Algorithm 1 to locate Java methods in memory. To be more specific, we examine the

program counter pc to check whether the current running function $func$ is either `ArtMethod::Invoke()` or `DoCall()`. If so, we further fetch the memory regions, mem_{method}, containing the compiled or interpreted Java method that is about to execute.

In the meantime, we intercept every memory write operation to obtain the addresses of modified memory regions. As illustrated in Algorithm 2, all the dirty memory regions are stored in $MemUD_{dirty}$, which is being continuously updated every time a memory write occurs (Ln.1). Then, Algorithm 2 detects unpacking activities by identifying mem_{method} for each basic block (Ln.12), and checking if the identified method falls into the dirty memory region (Ln.13). If that is true, unpacked code is discovered and we dump the method code and meta-data. After that, we also remove the mem_{method} from $MemUD_{dirty}$ (Ln.15), so that next time when the same method code is invoked, DROIDUNPACK will not count it as unpacked new code.

Note that this algorithm skips behaviors performed by Webview if JavaScript is enabled (Ln.7 to Ln.10). This filter is implemented to avoid potential false positive from JavaScript Just-in-time compilation (JIT) technique since its behavior can be mistakenly considered as packing.

Self-modifying code detection. Self-modifying code can be considered as a specific kind of unpacking. In addition to introducing decrypted new code, it also modifies the executable that has been previously launched. This practice, prevalently adopted by traditional PC malware, intends to cover the trace of historical execution or to change control flow and therefore needs special attentions.

To detect this, DROIDUNPACK searches particularly for the operation sequence $execute \rightsquigarrow (write \rightsquigarrow execute)$ conducted on the same memory region. Such a sequence indicates that a previously executed code region has been replaced by newly unpacked code. Algorithm 2 depicts the detection of self-modifying code as well. In addition to the aforementioned unpacking detection, this algorithm collects every executed basic-block region mem_{code} (Ln.11). The aggregation of all these code blocks, Mem_{code}, thus represents previously executed code (Ln.20). Hence, if mem_{method} is detected to be a newly unpacked method, the overlap between mem_{method} and Mem_{code} (Ln.16) demonstrates the presence of self-modification.

Multi-layer unpacking detection. Unpacking is not necessarily a one-time operation. If DROIDUNPACK realizes that multiple code sections have been unpacked gradually over time, it can reveal the existence of multi-layer unpacking. Concretely speaking, DROIDUNPACK considers all the continuously decrypted but not yet executed code belongs to the same unpacking layer, and therefore the execution of previously unpacked code indicates the end of a layer.

Algorithm 2 shows the detection details. First, we collect another copy of dirty memory $MemUL_{dirty}$, specifically for computing unpacking layer, again via observing memory writes (Ln.2). Then, we examine the overlap between the identified Java methods mem_{method} and $MemUL_{dirty}$ (Ln.21). A non-empty intersection, indicating an execution of dirty code region is about to happen, triggers the increment of $layer$ count

Algorithm 2 Analysis Using DROIDUNPACK

```
1:  MemUD_dirty ← {Overwritten memory regions updated by memory write monitor.}
2:  MemUL_dirty ← {Overwritten memory regions updated by memory write monitor.}
3:  layer ← 0
4:  Mem_code ← ∅
5:  for basic_block ∈ App execution trace do
6:      pc ← GETBEGINADDRESS(basic_block)
7:      mod ← GETCURRENTMODULE(pc)
8:      if JavaScript enabled and mod = "libwebview" then
9:          Continue
10:     end if
11:     mem_code ← GETADDRESSRANGE(basic_block)
12:     mem_method ← LOCATECODEINMEM(pc)
13:     if mem_method ∩ MemUD_dirty ≠ ∅ then
14:         DUMPMETHOD(mem_method)
15:         MemUD_dirty ← MemUD_dirty − mem_method
16:         if mem_method ∩ Mem_code ≠ ∅ then
17:             Self-modifying code is detected.
18:         end if
19:     end if
20:     Mem_code ← Mem_code ∪ mem_code
21:     if mem_method ∩ MemUL_dirty ≠ ∅ then
22:         layer ← layer + 1
23:         MemUL_dirty ← ∅
24:     end if
25: end for
    output layer as count of unpacking layers
```

(Ln.22) and eventually the accumulated count is provided as output. Once a new layer is discovered, we also clear the dirty memory $MemUL_{dirty}$ (Ln.23). This is to ensure that executing any unpacked code from the last layer does not cause DROIDUNPACK to increase the layer count.

Java native interface inspection. To avoid static inspection, sensitive APIs can be triggered through Java Native Interface (JNI) calls. Hidden bytecode or native code may also follow the same practice. Therefore, even if decrypted code has been captured, the static analysis of dumped code still may not successfully reveal the complete behavior of a packed app.

To make things even more complicated, packed apps can make recursive JNI calls. That is, a Java function $Func1$ can be invoked from a native function $Func2$ which is called through JNI from another Java function $Func3$. To handle cases like this, boundaries of each JNI call need to be captured.

Through the monitoring of context switching between Java and native modules, DROIDUNPACK can reliably detect the entrance and exit of each JNI calls and infer the boundaries. It further inspects all calls made at both Java and native side. In particular, DROIDUNPACK focuses on the detection of sensitive Android API calls invoked through JNI from native components. To identify sensitive API calls, we rely on the discovery of PScout [16].

E. Discussion

Data Compression and Encoding Techniques such as data compression/encoding are not considered as packing tech-

niques by DROIDUNPACK because they only introduce memory writes but do not execute at the same memory region. As a result, data compression and encoding will have no impact on our system.

Supporting Android versions. Being built upon DroidScope [39], DROIDUNPACK deliberately chooses to support Android 4.2 (DVM only) and 5.0 (DVM was replaced by ART) to cover the two runtime environments in Android. Supporting more Android versions will require relatively small efforts, such as recompiling the kernel and updating offsets for relevant data structures. Moreover, since variant versions of Android (e.g., Android Wear, Android Auto) share the same fundamental runtime environment, DROIDUNPACK should be able to support them with some fairly simple twists.

Emulation Detection. DROIDUNPACK is an emulation-based approach which means it cannot handle apps with emulation detection. To be more specific, our system cannot perform any automatic behavioral analysis if the apps hide all behaviors when they detect the existence of emulator. To deal with this limitation, DROIDUNPACK monitors four common anti-emulation techniques reported by SophosLabs [32] including examining services information, build information, system properties and the presence of emulator related files such as "/sys/qemu_trace". If any of the techniques is used by the testing app, DROIDUNPACK will raise alert which allows us to perform further manual investigation.

IV. STUDY METHODOLOGY

To answer the four sets of research questions brought up in Section I, our study of Android packer/unpacker follows a well-defined study methodology which consists of a broad range of automatic analysis using the capability of DROIDUNPACK as well as some manual investigations. This section elaborates on the methodology that we have systematically identified and itemized to facilitate the answers to each and every question.

A. Dataset and Setup

In order to accomplish the aforementioned tasks, we have gathered five datasets including:

- **Dataset 1**: We hand-pick seven popular and representative commercial packers including `Ali` [5], `apkprotect` [1], `baidu` [6][1], `Bangcle` [7], `ijiami` [9], `Qihoo` [11] and `Tencent` [12].
- **Dataset 2**: To study those commercial packers, we implement five representative apps, consider them as ground truth and perform diff analysis with their packed counterparts. To make sure we can seize modifications done by packers to majority of Android apps, the apps are designed to be concise yet still cover all four Android components - Activity, Service, Content Provider and Broadcast Receiver, also with two widely used features - dynamic class loading and JNI function calling. We then leverage packers in Dataset 1 to generate packed apps.
- **Dataset 3**: For the sake of studying packing techniques among wild malware, we manage to collect 93,910 Android malware from VirusTotal [14] which are labeled as malicious by at least 50% of all detectors with a wide time span from 2010 to 2015.
- **Dataset 4**: Five recent malicious apps including Android.Malware.at_plapk.a, Android.Troj.at_fonefee.b, candy_corn, braintest and ghostpush are collected from a public malware repository in github [10] to study malware detection of commercial packers. And for plagiarized apps, we manually insert empty Android activities into three most popular benchmark apps - Vellamo, Quadrant and AnTuTu and create three plagiarized apps.
- **Dataset 5**: Lastly, we collect three state-of-the-art Android unpackers that are published in mainstream academic and industry security conferences [40], [43], [33].

B. Methodology

For each set of research questions, we elaborate our methodology by listing four most important aspects: 1). dataset, 2). challenges and solutions, 3). detailed analysis and 4). limitations. Dataset section is to describe the data samples used for answering the specific set of questions. Challenges and solutions section is to list all the technical challenges to be addressed during the study as well as our proposed solutions. Detailed analysis section describes the proposed analysis to be performed in order to explore the answer. Limitations section is elaborated to discuss the possible limitations of our analysis.

Question set 1: Are Android packers (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware? What are the distributions of different commercial and custom packers across Android apps? How do the distributions change over time?

The first set of research questions is to understand the high-level landscape of current Android packers among malware, including the popularity of Android packers, distributions of each individual type of packers and how the distributions change over the years.

Dataset. In order to understand the high level landscape of Android packers, we utilize Dataset 3, the malware sample set which includes 93,910 samples in the wild with a wide time span from 2010 to 2015.

Challenges and solutions. There are two major challenges for conducting this study. First, understanding the existence of Android packers within malware samples is needed. Second, we have to further differentiate and recognize different types of packers. For the first challenge, we leverage the multi-layer unpacking detection capability in DROIDUNPACK to understand the existence of packing. As long as there exists a single layer of unpacking during the execution of a malware sample, we can then confirm the existence of packing within that sample. For the second challenge, as stated in [43], [40], commercial packers have strong and stable signatures across different versions. In our study, we rely on those signatures including activity names and native library names to identify the existence of different commercial packers. We collect signatures from packers in Dataset 1 and consider

[1] `baidu` packer requires Chinese ID so we exclude it in the detailed analysis

other packers as custom ones. Thanks to DROIDUNPACK, unlike previous works [43], [40], we are able to detect all the packers, commercial or custom, based on only intrinsic packing behaviors.

Analysis. To answer the first set of research questions, we first execute all malware samples in the dataset using DROIDUNPACK, during which we detect and record the existence and usage of different packers. We then calculate the ratio of packed malware among all the malware samples. Furthermore, we count the usage of each known packer and consider others as custom. Lastly, we extract the creation time for each sample and examine how the yearly distributions of different packers change from 2010 to 2015.

Limitations. Our analysis has several limitations. First, since we only collect signatures for the six packers, which are by no means complete, the ratio for the custom packers may be overestimated. Second, theoretically, the custom packers can impersonate the commercial packers by using the same signatures. However, we argue that the six packers are popular and representative. And despite the fact that the custom packers can impersonate the commercial packers, they probably do not have enough incentive to do so.

Question set 2: How do Android packers work? Is it very different from traditional packing? What are the security impacts when applying Android packers to apps? Is it easy for malicious developers to exploit commercial packers and pack their malware or plagiarized apps?

The second set of research questions is about detailed behaviors and impacts of Android packers.

Dataset. To understand the detailed behaviors and impacts of Android packers, we need to have ground truth first. To this end, we make use of Dataset 2 to conduct our study. We further leverage Dataset 4 to study the malware and plagiarism defense of commercial packers.

Challenges and solutions. Two major challenges need to be resolved here. First, we need to separate the behaviors of packer's code from the original code. The second challenge is to fully understand the detailed behaviors of Android packers at different levels including Java level, native level and their interactions via JNI. For the first challenge, we run our benign apps along with their packed counterparts under DROIDUNPACK and record all the behaviors. Then we perform diff analysis to reveal only the behaviors of packer's code. The second challenge requires us to understand the behaviors at different levels. For Java level behaviors, we rely on hidden code extractor in DROIDUNPACK to extract packed DEX code and further perform static analysis using other tools such as FlowDroid [15]. For native level behaviors, we are able to retrieve OS-level view and leverage self-modifying code detector and multi-layer unpacking detector from DROIDUNPACK to observe the unpacking behaviors. Moreover, we intercept important function calls to trace file operations, memory mapping and more to uncover how code is unpacked and loaded into the memory. For JNI interactions, JNI inspector in DROIDUNPACK is utilized to monitor everything that happens through JNI, especially sensitive API calls.

Analysis. In order to grasp how Android packers work, we first execute and record all the behaviors of packed apps and compare with ground truth. Then, manual investigation is performed on top of the behaviors to further understand the semantics and underlying rationale behind those behaviors so that we can not only know what happens but also why it happens. For the sake of understanding security impacts of commercial packers, we first extract the hidden code using DROIDUNPACK and examine the packer added code via static analysis tools and manual investigation. Lastly, we act like malicious developers to submit malware samples and plagiarized apps to commercial packing services and check whether the submissions can be detected and prevented. To further measure the impact of packing in terms of malware detection, we submit the packed malware samples to VirusTotal [14].

Limitations. Since our analysis involves human effort to investigate the behaviors, there may be some behaviors that fail to catch our attention, therefore are missed by our study. Also, we fail to find any service that could allow us to measure the impact of packing in terms of plagiarism detection.

Question set 3: Have Android packers been evolving and how? And what are the future trends of this evolution?

The third set is a two-part question. It is related to the evolution of Android packers which is extremely important for learning the current status as well as forecasting the future trend.

Dataset. Evolution can only be observed via analyzing large amount of data. Thus, we use all samples including Dataset 2 and 3 for this purpose.

Challenges and solutions. One challenge here is how to define evolution. We define it as the change of complexity during unpacking process and characterize this complexity in two aspects: the number of unpacking layers and some unique behaviors that are designed to defeat existing unpackers. Inevitably, packed apps have to perform an unpacking process before original code can be executed. This unpacking process is not necessarily a one-time effort, in stead, it may contain multiple layers of packing and unpacking. Subsequently,the number of unpacking layers can be a quite representative attribute to measure the complexity of packers. Furthermore, we also propose to use new behaviors that are only discovered in recent years as a sign of evolution as well.

Analysis. To capture the Android packer evolution, we first consider the number of unpacking layers by executing all the packed malware samples and utilize the multi-layer unpacking detector in DROIDUNPACK to collect the layers distribution information over different years. We hope to see a clear trend of increasing layers of packing. Then, we scrutinize some novel behaviors captured by DROIDUNPACK that are clearly targeting unpackers and only appear in the recent years and then use those to demonstrate the evolution.

Limitations. Our current measurement of complexity is by no means comprehensive and complete, as compared to the one used for measuring the traditional PC packers [36]. We leave a more comprehensive study of complexity and evolution as future work.

Question set 4: How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?

The last set of questions is about current Android unpackers. Due to the aforementioned complexity of Android packers, we would like to see if state-of-the-art Android unpackers can handle all the cases correctly from a design point of view.

Dataset. We utilize Dataset 5, a group of state-of-the-art Android unpackers to understand the internals of unpackers and their fundamental design limitations. To test the effectiveness, we propose to use the samples in Dataset 2 and some malware with advanced behaviors from Dataset 3 to evaluate those unpackers.

Challenges and solutions. The major challenge is to understand the designs and fundamental limitations of current Android unpackers. The solution for this challenge is to study through the literatures and the source code in order to fully understand those unpackers.

Analysis. By reviewing the literatures and source code, we perform manual analysis on the fundamental designs and limitations of each unpacker. To better understand the whole picture of current Android unpackers, we would like to conduct experiments and further compare them with DROIDUNPACK.

Limitations. Although the three Android unpackers are state-of-the-art tools, there may exist other tools that embrace unique designs and share different insights. We will continue this investigation in our future research.

V. OUR FINDINGS

In this section, we present our answers to the four sets of questions raised earlier.

A. Question Set 1: High-level Landscape

As discussed in the previous reports [8], [4], researchers have found that malware samples have been leveraging packing techniques to evade detections and infiltrate into Android ecosystem. Therefore, understanding the high level landscape of packing techniques among Android malware samples has become the very first thing for us to study.

Question 1.1: Are Android packers (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware?

Answer: Yes, Android packers are being abused by malware author and packing techniques are quite prevalent among malware. We present Finding 1 to further answer the question.

Finding 1. Malicious developers extensively leverage packing techniques to hide malice. By depicting the yearly distribution for packed apps, Figure 2 shows the fact that packing techniques have embraced similar popularity among malware from 2010 to 2015 with an average of 13.89%. Interestingly, this observation contradicts AppSpear [40] where the authors claim that the ratio of packed apps is increasing. The reason to this discrepancy is that AppSpear only detects packers by signatures thus misses custom packers while we are able to extract a more complete picture of packing techniques.

Question 1.2: What are the distributions of different commercial and custom packers across Android apps?

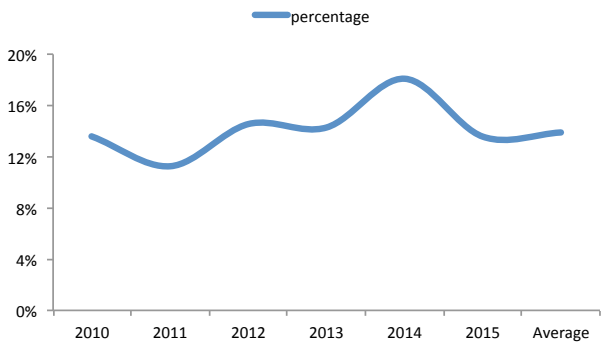

Fig. 2: Yearly distribution.

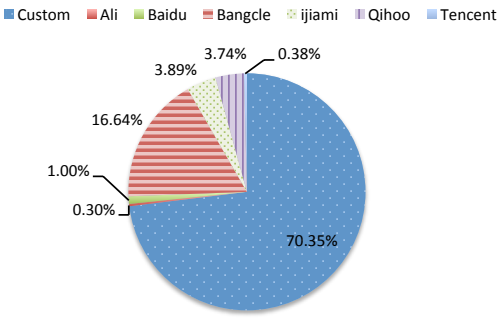

Fig. 3: Packer distribution.

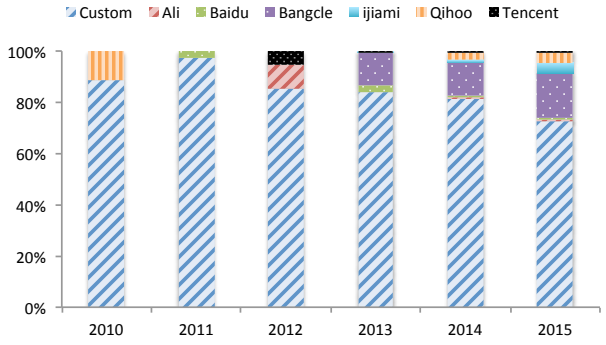

Fig. 4: Trend of packer distribution.

Answer: The following finding 2 answers this question by showing the distributions of different packers.

Finding 2. Custom packed malware samples take up the largest portion of all packed malware. The distribution of packed app over different packers is presented in Figure 3. For all the 93,910 malware we collect, 13,052 (13.89%) of them are packed. We find custom packing takes up the biggest portion (70.35%) of the packed malware and followed by Bangcle which is utilized by 16.64% of the packed malicious apps. This finding further indicates the necessity of DROIDUNPACK as no existing tool can analyze custom Android packers.

Question 1.3: How do the distributions change over time?

Answer: By depicting the trend of packer distribution from

2010 to 2015, Finding 3 gives us the answer to the above question.

Finding 3. Android commercial packers are increasingly abused by malware. Apart from what has been stated above, Figure 4 takes one step further to illustrate the trend of packer distribution among different years. Clearly, commercial packers are increasingly leveraged by malware as the ratio of custom packers has decreased gradually from 88% in 2010 to 69.3% in 2015. This finding is then on par with what AppSpear [40] claims.

Summary for finding 1-3. Two observations can be made from the above study. First, the existence of packed malware is a real threat with very long history tracing back to early stage of Android. This indicates that the study of Android packing techniques is not only beneficial but also necessary for malware analysis. Second, while custom packers still dominate, commercial packers are gaining popularity steadily over time. Despite the effort of enforcing different kinds of detection techniques, commercial packers still have a long way to go for filtering out malware from being packed.

B. Question set 2: Detailed Analysis on Android Packers

From the previous results, we know Android packers including commercial and custom ones have been widely abused by malware. It is important to understand the behaviors of these packers, especially the unique behaviors that do not appear in the traditional PC packers. To this end, we perform detailed analyses on both commercial and custom packers. Our study shows Android packers have embraced some unique packing techniques that are not reported by the previous Android and traditional packer research [36], [43], [40]. More importantly, as free services, we find commercial packers are not as secure and innocent as they claim to be.

Question 2.1: How do Android packers work? Is it very different from traditional packing?

Answer: Yes, Android packers are indeed very different from traditional packers. We elaborate the differences using Finding 4.

Finding 4. Commercial packers have adopted many unique yet unreported features for anti-unpacking. Following the methodology described in Section IV-B, we comprehensively study the behaviors of six popular commercial packers. Table I summaries unique features of those packers.

App context restoration via JNI. Application context restoration is a common practice among Android packers. To hide the original code completely, packers including `ijiami`, `Qihoo` and `Tecent` create their own wrapper applications. These wrapper applications collect environment information (e.g. CPU architecture), load necessary libraries, unpack the original code and restore the app context back to the original code. `AttachBaseContext()` is the function that packers usually override to perform these tasks since it is called by the framework even before `OnCreate()` and has the ideal timing for pre-processing. JNI, on the other hand, is extensively used by packers for various of reasons. First, JNI functions which are declared within Java level but defined in native libraries will break the control-flow and data-flow analyses. Second, some functions with heavy computations can be written as native yet still be called from Java level to boost performance. Third, packers can also leverage JNI to hide sensitive behaviors from being detected, thwarting most of the current Android app analyses. By leveraging DROIDUNPACK JNI analysis capability described in Section III-D, we can bridge the gap between Java and native, thus understanding exactly what has happened at native level and how Java and native codes cooperate. In our study, we utilize PScout [16] and DROIDUNPACK to monitor sensitive API calls within JNI and discover that only `ijiami` packer cleverly invokes its application context restoration via JNI, making it harder to be detected.

Native/DEX obfuscation. As reported by the previous work [43], [40], obfuscation techniques are widely employed by commercial packers at both DEX and native levels. DEX code level obfuscation includes a wide range of techniques, such as string obfuscation, reflection, dead code injection and more. But for native code, things are slightly different. In Android, JNI builds up a bridge between Java code and native code, allowing them to interact with each other. There are mainly two ways of performing method lookup: 1) traditionally, developers could name the JNI functions in a specific way using `Java + package name + class name + function name` format so that the function mapping is automatically handled; 2) JNI functions can also be explicitly registered via `JNI_OnLoad`. All the packers other than `apkprotect` take this approach so that they can randomize function names, making it more difficult to obtain the control flow graph. Moreover, most of the commercial packers will introduce native libraries as stated in [43] for the purpose of performing unpacking. We discover that all of the native libraries are equipped with encryption to data and code sections within binary so as to prevent analysis. At runtime, the libraries are first loaded into memory, then unpacking code will identify the address by reading from `/proc/pid/maps` and decrypt the libraries dynamically. This kind of behavior is observed using DROIDUNPACK through libc function interception and memory operation analysis.

Multi-layer unpacking. Many Android unpackers [43], [40], [34] depend on an assumption that there exists a clear boundary between packer's code and original code within packed apps to function normally. However, according to our observation, this assumption no longer holds. According to our study, many commercial packers turn to multi-layer unpacking strategy, meaning other than unpacking the original code at once and loading into memory, they unpack the original code layer by layer during execution. This technique will obviously render the current memory dump based unpackers useless since the dumped memory will contain mostly the unreadable packed code other than the original code. Table II shows that `Bangcle`, `ijiami`, `Qihoo` and `Tencent` adopt this unpacking technique, among which, `Tencent` is the most complex one.

Pre-compilation. Code in OAT file is allowed to be compiled into native code or remains as DEX. From the app analysis point of view, DEX code is much better than native code in terms of readability and simplicity as it preserves semantic meaning of the program. So willfully, packers want to avoid revealing DEX code as much as possible. However, as

TABLE I: Commercial packer behavior.

	apkprotect	Ali	Bangcle	ijiami	Qihoo	Tencent
Context switching via JNI	✗	✗	✗	✓	✗	✗
Native/DEX obfuscation	✓	✓	✓	✓	✓	✓
Pre-compilation	✗	✗	✗	✗	✓	✗
Multi-layer unpacking	✗	✗	✓	✓	✓	✓
libc.so hooking	✗	✗	✓	✗	✗	✗
Self modification	✗	✗	✗	✗	✗	✗
Component hijacking vulnerability	✗	✗	✗	✗	✓	✗
Information leakage	✗	✗	✗	✗	✗	✓

TABLE II: Multi-layer unpacking.

	# of layers
apkprotect	1
Ali	1
Bangcle	9
ijiami	4
Qihoo	4
Tencent	40

we find out in the study, completely transforming original app's DEX code into native code is such a challenging idea that all packers simply avoid. Nevertheless, we would like to see if any of the packer's code is pre-compiled into native even before the installation. In order to detect this behavior, we first configure the `dex2oat` (the ART compiler) to be `interpret-only` so that theoretically no code should be compiled into native code at all. Then, we utilize DROIDUNPACK to extract hidden code from all the packed samples and check if there still exists any native code. While the answer is expected to be negative, we actually find that the sample packed by `Qihoo` packer has pre-compiled DEX code. Further looking into it gives us more details. Just like most of the packers, when packing with `Qihoo`, the packer will insert a few new components into the app. However, unlike other packers, it pre-compiles some of the packer-added DEX code into native code by invoking `dex2oat` with default configuration, ignoring the `interpret-only` flag. This technique is much less difficult than converting app's original code into native code, but can still be very useful to hinder analysis tools understanding the whole picture, especially for tools that hook Android runtime functions, e.g., DexHunter [43].

libc.so function hooking. Among many anti-debugging techniques that the packers adopt, libc modification is a very special one. We only observe this behavior with `Bangcle` packer. By analyzing memory operations, we discover that it actively modifies the libc.so module. Further inspection shows the packer tries to hook a series of important libc functions such as `read`, `write`, `open`, `mmap`, etc. This hooking behavior will disrupt many unpackers. Unpackers such as DexHunter [43] rely on libc functions like `fwrite` to dump the code from memory into files. When using these packers to unpack `Bangcle`, the app will simply crash if these libc functions are called, therefore, completely breaks the unpacking process. In order to bypass this restriction, one has to modify the unpackers and directly invoke the associated system calls instead of libc functions. This requirement certainly puts an extra hurdle for unpacker users. This technique, on the other hand, will not affect DROIDUNPACK since it is based on whole-system emulation.

Question 2.2: What are the security impacts when applying Android packers to apps?

Answer: We discover severe security vulnerability and data breach[2] that some commercial packers are responsible for.

Finding 5. Android packers have led to severe security vulnerability and data breach affecting more than 1 billion users.

Commercial packers are believed by developers to be secure and only to protect intellectual property. The results of our study, however, shows that by applying some of the packers, the apps will have serious component hijacking vulnerability as well as information leakage problem.

Component hijacking vulnerability. Component hijacking vulnerability in Android is dangerous due to the fact that it allows malicious app to invoke vulnerable components and achieve a series of goals including privilege escalation and information stealing. One component within Android app can be considered as a potential target as long as its attribute "android:exported" is set to true in Manifest file. During the study, surprisingly, we notice two potential vulnerable components created by `Qihoo` packer: a content provider and a service. By examining the Manifest file, we find that the attribute "android:exported" for both components are set to be true, indicating the possibility of component hijacking vulnerabilities. Further study shows that the service is successfully launched during app execution. Since the service is fully packed, we utilize DROIDUNPACK to extract the hidden code and conduct a thorough investigation. Eventually, we confirm that the service is indeed vulnerable to component hijacking attacks. The service handles two different intents, one of them allows the service to download files from remote server and replace arbitrary file within the app using the app's permission. We manage to write a Proof-of-Concept code that can exploit this vulnerability by downloading a DEX file from our own server and replacing arbitrary files within the vulnerable apps. In a nutshell, using this packer to pack a perfectly secure app exposes serious arbitrary file write and even arbitrary code execution. We have reported this security issue, it was immediately acknowledged and assigned highest priority.

Information leakage. Our study unveils another astonishing fact that one of commercial packers adds code to the original

[2]This issue was identified by static analysis. We tried to contact *Tencent* to confirm but no reply so far.

app to collect sensitive user data and send back to its own servers, thus causes an information leakage problem. As shown in Table I, among the packers we study, `Tencent` packer introduces this kind of dangerous behavior. Upon packing, it will add six new permission requests to the original apps including some very sensitive ones such as `ACCESS_NETWORK_STATE` and `READ_PHONE_STATE`. Once the packed app is launched, it will collect sensitive user data such as "deviceId", "subscriberId", "MAC address", and send them back to its own server via an insecure HTTP connection. This behavior not only leaks user sensitive information to their server without any user awareness but also gets them exposed to the public as attackers can easily eavesdrop via man-in-the-middle attack. During the investigation, we rely on DROIDUNPACK to extract hidden code and discover the information leakage using FlowDroid [15], a state-of-the-art context-, flow-, field-, object-sensitive static analysis tool.

Impact. By simply examining the apps that are using the two problematic packers, we can draw a conclusion that these two security issues are very severe as they are affecting more than 1 billion users right now. `Qihoo` packer, which introduces component hijacking vulnerability, has been leveraged by some most famous apps including Gaode Navi, Qianniuniu Finance. Gaode Navi is actively used by more than 500 million users as their daily navigation app. The vulnerability within it can easily be leveraged by attackers to obtain users' daily routing information. Qianniuniu finance, which has been downloaded for more than 3 million times, is an investment app. The vulnerability within it can severely damage users' financial security. The information leakage issue, on the other hand, is affecting even more users as it is applied by a series of popular apps including QQ, a chatting app that has more than 800 million active users.

Question 2.3: Is it easy for malicious developers to exploit commercial services to pack their malware or plagiarized apps?

Answer: Yes, Finding 6 shows that it is very easy for malicious developers to exploit commercial packers and avoid being detected.

Finding 6. Malicious developers can easily exploit commercial packers to pack malware and plagiarized apps and thus evade detections.

As we know, all of the packers except for `apkprotect` provide on-line services which aim to present packing service to protect developers' intellectual property while avoid being leveraged by malware and plagiarized apps. Consequently, they all claim to implement some kinds of security scrutiny. To this end, we conduct a study on this subject by submitting 5 confirmed recent (early 2016) malicious apps and 3 plagiarized apps to these packers and the results are presented in Table III.

Malware defense. Malware detection is hard but packed malware detection is even harder [8], [2]. To protect Android users from being compromised, all studied commercial packing services claim to conduct advanced code analysis on submitted code to rule out malware. However, our study result somehow shows otherwise. Among those five packers, `Qihoo` is namely the best when it comes to malware defense. It detected our first malware and blocked us from further submission. `Ali` also managed to detect all five malicious apps and prevented us from packing them. Together with Figure 4, we can clearly observe a huge improvement over malware detection for `Qihoo` and `ALi`. Other packers, however, can only detect a portion of them resulting in successful packed malware. We then submit the original malware samples as well as the packed ones to VirusTotal [14]. As illustrated in Table IV, the detection rates for malware have dropped significantly after packing, depicting the fact that malicious developers can easily exploit commercial packers to pack their malware and evade detection.

Plagiarism detection. Although all packers claim to help developers scan over multiple Android markets to detect plagiarism, no one actually stops developers from submitting plagiarized apps to its server. In our study, we submit three plagiarized apps to those packers and easily create packed plagiarized ones through all packers without any issue. This security loophole can be effortlessly leveraged by plagiarized app developers to pack their apps, rendering plagiarism detection more difficult.

C. Question set 3: Evolution of Android Packers

Question 3.1: Have Android packers been evolving and how? What are the future trends of this evolution?

Answer: Yes, Android packers are clearly evolving. We describe this trend with Finding 7.

Finding 7. Android packers have been evolving very fast in the last few years. Based on the systematic study of large number of packed malware samples over multiple years, we observe that Android packers are clearly evolving. We characterize this evolution in two different aspects: the number of unpacking layers and featured behaviors.

Fig. 5: Layer distribution.

Number of unpacking layers. We have seen multi-layer unpacking in commercial packers, but we haven't seen such complicated unpacking process as shown in Figure 5. In year 2015, there exist about 1.3% of custom packed Android malware that unpack their hidden code with more than 1000 layers. This level of complication is never observed in commercial packers and certainly brings tremendous difficulty for unpackers to operate. In contrast, the most complicated custom packed malware we have in year 2010 has only 6 layers. The ratio of packed malware that equip with 10 or more layers unpacking has grown from 0% in 2010 to 24.73% in 2015 which is a clear indicator that Android custom packers have been evolving in a fast pace.

TABLE III: Security scrutiny.

	apkprotect[1]	Ali	Bangcle	ijiami	Qihoo	Tencent
Malware defense failure	5/5	0/5	2/5	2/5	0/5[2]	1/5
Plagiarism detection failure	3/3	3/3	3/3	3/3	3/3	3/3

[1] apkprotect is not on-line service and has no prevention for malware or plagiarism.
[2] Qihoo detected first attempt and blocked further malware submission.

TABLE IV: Malware detection rate comparison.

Malware name	Original detection rate	Packed detection rate
Android.Malware.at_plapk.a	61.67%	26.67%
Android.Troj.at_fonefee.b	66.67%	35%
braintest	63.33%	37.29%
ghostpush*	70%	N/A
candy_corn	68.33%	38.98%

* All commercial packers can successfully detect it as malware.

Behaviors. We consider two interesting behaviors as a clear sign of evolution for Android packers. First is the aforementioned libc.so hooking. As described, `Bangcle` packer modifies libc.so module so as to hook functions and prevent unpackers such as DexHunter [43]. By analyzing the timing, we can see this behavior was not added by `Bangcle` until DexHunter has released. Clearly, `Bangcle` itself is evolving to defeat unpackers. Second, a more advanced technique has been observed by us that it modifies DEX code at runtime so that apps can change their behaviors dynamically. By closely monitoring memory writes and code executions as described in Section IV-B, we observe this behavior in 20 out of 93,910 wild malware samples, 6 from 2014 and 14 from 2015. Self modification is normally done via JNI since native code is more suitable than Java code for memory manipulations. The app invokes JNI function which is responsible for code modification to start this process. The function first finds the right module by scanning over `/proc/self/maps` file which stores the addresses of all modules. Then, it locates OAT file in memory via magic number "oat\n" and parses the OAT file to acquire the correct class and method to modify. Before modification can be performed, it needs to change the memory protection by calling `mprotect` to make it writable. Finally, payload is inserted into the designated code region via `memcpy` and gets executed. This kind of technique has been useful for hiding sensitive code from static analysis and unpacking tools. For example, one sample dynamically modifies the code so that other than invoking the original function, it calls a different one. Note that this technique is designed to work on DEX code which means ART may have compatibility issue as the compiler compiles DEX code into native code Ahead-Of-Time. To verify, we test those apps again with `dex2oat` configured as "speed" mode and observe that self-modifying behavior has disappeared.

Future trends of this evolution. Android packers are evolving. We believe the future Android packing technique could push its limits further into several directions that could get unpacking increasingly problematic. First, more interactions between DEX code and native code will appear in packing techniques. Native code is favorable for packers as it is unobservable from Java level, and thus is more difficult to extract. Moreover, it is a known challenge to recover semantics information even if unpackers can successfully extract the code. The pre-compilation behavior we observe is only the very first step that falls into this category. Second, Android packing strategy will continuously become more sophisticated. We have seen Android packers growing from single-layer to multi-layer and will probably see packers carrying other features as what has happened in PC packer [36], such as cyclic transition, multi-frame and more. Third, Android packers may eventually turn to emulation-based packing techniques which is more advanced and can defeat all existing unpackers including DROIDUNPACK.

D. Question set 4: Android Unpackers

We study the designs, implementations and limitations of the mainstream Android unpackers and test them against the packed malware samples in the wild.

Question 4.1: How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?

Answer: No, state-of-the-art Android unpackers are not working properly as expected. We clarify this answer by introducing Finding 8 which gives an overview of how those unpackers perform.

Finding 8. State-of-the-art Android unpackers have serious design limitations that they cannot handle advanced Android packers. Current unpackers could be roughly categorized into three types based on distinct system designs. 1) Locate DEX file by signature and perform memory dump; 2) Modify DVM to hook certain important functions to find DEX file and then dump the code; 3) Modify DVM to dump Dalvik data structures on the air and then assemble them back into a DEX file. As discussed in Section IV-B, we pick three state-of-the-art unpackers from each category and compare them with DROIDUNPACK in Table V.

Design choices. Kisskiss [33] follows a very traditional unpacking process. It is compiled as a stand-alone program and pushed into Android system for attaching to and accessing memory of target application using `ptrace`. It recognizes `odex` objects based on the memory map and the magic number

and finally performs the memory dump. DexHunter [43], on the other hand, is designed to be more general-purpose based on a study of protections of current packers. Relying on customization of class loading of both Dalvik and ART runtime, it guarantees that all classes of `odex` are initially loaded, correctly located and then extracted. Certainly, this runtime customization approach is immune to anti-debugging and anti-emulation techniques. AppSpear [40] adopts techniques of bytecode extraction and DEX reassembling based on Dalvik instrumentation. Once the main activity is interpreted or a new DEX file is loaded, AppSpear extracts the inner Dalvik Data Structure (DDS) and performs a reassembling process to recover the DEX file. DROIDUNPACK takes a completely different approach from those unpackers by leveraging the whole-system emulation technique. It detects a packed app via monitoring program execution on overwritten code regions and relies on only intrinsic characteristics of Android runtime and enables VMI to recover hidden code.

Limitations. Unlike DROIDUNPACK, none of the existing Android unpackers can have a whole view in multiple levels of the system nor can they detect unknown packers in a complete fashion.

Besides this, Kisskiss faces several severe limitations. First, commercial packers usually deploy techniques, like anti-debugging or in-memory obfuscation towards `odex` objects, to defeat this unpacking process [43]. Since Kisskiss relies on the magic number to dump `odex` objects, it does not work with unknown new packers or even the upgraded version of existing packers. Moreover, as it only dumps the memory once based on signature and could be easily defeated by more advanced techniques such as multi-layer unpacking and self-modifying code.

DexHunter mainly improves the way of locating DEX file in memory by hooking class loading functions. This design choice makes it more robust than Kisskiss. However, it is still far from being perfect. First of all, as stated in the previous section, libc.so function hooking in Bangcle packer could defeat DexHunter unless users modify it accordingly. Second, multi-layer unpacking and self-modifying code will result in incomplete and even erroneous code dump because DexHunter only dumps the memory at the class loading time when the hooking functions get triggered.

AppSpear customizes DVM to collect the DSS data structure so that the code it dumped must be unpacked. However, it still exposes a few important limitations. First, it only works in Dalvik but not in the latest ART. In ART, code can be compiled into native during installation and will not even appear in any Dalvik data structures. Second, finding correct timing to extract DSS can be a very challenging task. By default, it only considers the main activity as unpacking point [40] which may lead to incomplete code coverage.

Despite the fact that DROIDUNPACK can overcome limitations described above, it does have a few drawbacks. As shared by all dynamic analysis techniques, DROIDUNPACK certainly suffers from limited cover coverage as it can only dump the code that executes. And since it is built on top of whole-system emulation, packers that enforce anti-emulation techniques will inevitably break the analysis.

Experiments. We conduct the experiments upon Android 4.3 and 4.4 emulators for two popular open sourced unpackers. DexHunter and Kisskiss with two datasets. 1) Dataset 2 in Section IV-A; 2) self-modifying malware samples collected in the above study. As shown in Table V, at the time our experiment was carried out, Kisskiss failed to dump memory from all six packers. This is probably because the signatures that Kisskiss relies on have been changed.

The experiment results then show that DexHunter is rather sensitive in the arms race with packers. The prototype relies on a fingerprint (feature string) of each known packer, which we found only works for `Tencent` packer now. Note that the result for `Tencent` is still incomplete as it adopts multi-layer unpacking.

VI. RELATED WORK

Packing & Unpacking techniques. Runtime packers have been well-studied in the traditional context of desktop operating systems. Polyunpack [30] performed a differential analysis between statically disassembled binary code and its execution trace, and observed the difference to discover potential hidden code. Renovo [27] applied whole-system emulation to unpacking and was able to trace program execution at instruction level. It instrumented memory writes and execution to detect unpacked code. Besides, it can address multi-layer unpacking and create code dumps for every detected layer. Omniunpack [29] also monitored memory writes and execution but at the page granularity. It aimed to provide an efficient and resilient unpacker and therefore relied on the memory protection mechanisms in operating system to enable detection. Similarly, Eureka [31] also conducted coarse-grained packer analysis. It first monitored system calls to search for written and executed memory and then depended upon heuristics to identify and extract decrypted code.

Recently, Android packers have attracted the attention from both industry and academia. Kisskiss [33] leveraged ptrace to access application memory and searched for hidden DEX code based upon heuristics. DexHunter [43], to defeat anti-debugging mechanism utilized by packers, instrumented the class loading functions in Dalvik VM and ART runtime. Thus, it can extract any class that has been loaded for execution. AppSpear [40] took a step further and stitched all the discovered hidden classes together to reassemble a complete program.

The research community has also made efforts to systematically survey existing packing techniques. Bayer et al. [17] studied the off-the-shelf packers that were often used by malware authors. Ugarte-Pedrero et al. [36] presented a fairly comprehensive measurement in order to understand the complexity of packers. As a comparison, our study focuses on the interpretation of evolving packers and unpacking techniques that have adapted to the novel Android context. To this end, we have developed a new machinery to address the unique challenges in analyzing packed Android apps.

Android application analysis. Many previous efforts were made to pursue in-depth analysis of application behaviors. Ded [22], DroidSIFT [41], CHEX [28], PEG [18], FlowDroid [15], DroidSafe [24] and AppAudit [38] practiced static dataflow analysis to identify specific code (e.g. malicious code or heavy computation code [20]) in Android apps. TaintDroid [21], DroidScope [39], CopperDroid [35] and VetDroid [42] conducted dynamic taint analysis to detect

TABLE V: Study of unpackers

Tool	Design	Limitations	Open source	Recover code from commercial packers	Unpack self modifying samples
DexHunter	Modify DVM and hook class loading functions for locating and extracting DEX file	a) rely on feature string, which could vary when a packer is upgraded; b) difficult to find the right timing, can't deal with incremental packer, which means there isn't a single moment when all codes coexist in memory together	Yes	Success: Tencent. Failure: Ali, Bangcle, ijami, Qihoo. Not support: apkprotect	No
AppSpear	When MainActivity is launched or a new DEX file is loaded, AppSpear extracts inner DDS and reassembles the DEX file.	a) lack of support for ART; b) hard to find correct timing for extraction	No	N/A	N/A
Kisskiss	Use ptrace to attach to the memory of target applications, and identifies and dump odex objects based on memory map and magic number.	a) can't handle apps with anti-debug or in memory obfuscation techniques; b) requires understanding of the specific packer to get magic number thus doesn't work with unknown packer or even slightly upgraded existing packer	Yes	Success: none. Failure: all samples. Can find odex file in memory map, but failed in locating the correct address. So, pread syscall failed	No
DROIDUNPACK	Monitor program execution and memory operations based on whole-system emulation	a) cannot handle packed samples with anti-emulation	Yes	Success: Ali, apkprotect, Bangcle, ijiami, Tencent, Qihoo	Yes

suspicious behaviors at runtime. Static analysis requires access to complete bytecode program and therefore can be simply evaded by runtime packers. In contrast, dynamic analysis tools can potentially facilitate packer detection. Nevertheless, existing generic analysis frameworks cannot be directly applied to the identification, diagnosis and study of packed apps due to the lack of specific analysis capability.

VII. CONCLUDING REMARKS

In this paper, we conduct a comprehensive study on 6 major commercial packers, 13,566 packed malware samples out of 93,910 Android malware and 3 existing state-of-the-art unpackers in order to better understand the security issues. To facilitate our study, we develop DROIDUNPACK, a whole-system emulation based Android unpacker which can precisely recover hidden code. Our study has revealed that commercial packing services have not only been misused to encrypt malicious or plagiarized contents but also introduced various severe vulnerabilities to apps being packed. We have found evidence to demonstrate the prevalence and rapid evolution of custom packers used by malware authors. Unfortunately, we have also realized that current defense techniques often fall short due to fundamental design limitations.

Final thoughts. DROIDUNPACK is by no means the end of the game but merely a start for future endeavors as the war between packing and unpacking on Android continues. The real problem lies within the design choice of Android system. Unlike iOS which enforces code signing [13] to prohibit app from modification since it was last signed, Android allows the code to be modified even after installation. This feature opens a broad surface for Android packers to perform all kinds of packing techniques without any constraint. Granted, packers are also utilized extensively in legitimate ways for the purpose of protecting intellectual property. However, from the study we surely see packing techniques are currently abused by malware authors, exposing great threats to end users. This situation deserves more thinking for the whole community from a design point of view.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the National Science Foundation under grant 1664315 and DARPA under grant FA8750-16-C-0044. The IU authors are supported in part by the NSF CNS-1527141, 1618493, ARO W911NF1610127 and Samsung gift fund. We would also like to thank VirusTotal for granting us the privilege for the large scale query and app downloading.

REFERENCES

[1] "apkprotect," https://sourceforge.net/projects/apkprotect/, 2013.

[2] "McAfee Labs Threats report Fourth Quarter 2013," http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2013.pdf, 2013.

[3] "Android developers blog," https://android-developers.googleblog.com/2016/05/whats-new-in-google-play-at-io-2016.html, 2016.

[4] "ValerySoftware McAfee," https://securingtomorrow.mcafee.com/mcafee-labs/obfuscated-malware-discovered-google-play/, 2016.

[5] "alibaba," http://jaq.alibaba.com/, 2017.

[6] "Baidu," http://app.baidu.com/, 2017.

[7] "Bangcle," https://dev.bangcle.com/, 2017.

[8] "Charger Malware," http://blog.checkpoint.com/2017/01/24/charger-malware/, 2017.

[9] "ijiami," http://www.ijiami.cn/, 2017.

[10] "Malware repository," https://github.com/ashishb/android-malware, 2017.

[11] "Qihoo," http://jiagu.360.cn/, 2017.

[12] "Tencent," http://legu.qcloud.com/, 2017.

[13] "IOS code signing," https://developer.apple.com/support/code-signing/, 2017.

[14] "VIRUSTOTAL," https://www.virustotal.com/, 2017.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, June 2014.

[16] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[17] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, ser. LEET'09, 2009.

[18] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[19] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th international conference on Information security*, Berlin, Heidelberg, 2011.

[20] Y. Duan, M. Zhang, H. Yin, and Y. Tang, "Privacy-Preserving Offloading of Mobile App to the Public Cloud," in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, 2015.

[21] W. Enck, P. Gilbert, 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," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[22] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[23] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[24] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[25] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[26] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of CCS*, 2011.

[27] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, ser. WORM '07, 2007.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[29] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[30] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[31] M. I. Sharif, V. Yegneswaran, H. Sadi, P. A. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.

[32] SophosLabs, "Anti-emulation techniques," https://news.sophos.com/en-us/2017/04/13/android-malware-anti-emulation-techniques/, 2017.

[33] T. Strazzere, "Android hacker protection level 0," https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDATED.pdf, 2014.

[34] ——, "android-unpacker," https://github.com/strazzere/android-unpacker, 2015.

[35] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[36] 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 *SSP 2015, IEEE Symposium on Security and Privacy, May 18-20, 2015, San Jose, CA, USA*, 2015.

[37] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps." in *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)*, Scottsdale, AZ, November 2014.

[38] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland'15)*, 2015.

[39] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[40] 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 *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*, 2015.

[41] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs," in *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)*, Scottsdale, AZ, November 2014.

[42] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, November 2013.

[43] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, 2015.

[44] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)*, May 2012.

[45] ——, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[46] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.