What Did You Pack in My App? A Systematic Analysis of Commercial Android Packers

Zikan Dong
Beijing University of Posts and Telecommunications
Beijing, China

Hongxuan Liu*
Peking University
Beijing, China

Liu Wang
Beijing University of Posts and Telecommunications
Beijing, China

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China

Yao Guo
Peking University
Beijing, China

Guoai Xu
Beijing University of Posts and Telecommunications
Beijing, China

Xusheng Xiao
Case Western Reserve University
Cleveland, United States

Haoyu Wang†
School of CSE, Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Commercial Android packers have been widely used by developers as a way to protect their apps from being tampered with. However, app packer is usually provided as an online service developed by security vendors, and the packed apps are well protected. It is thus hard to know what exactly is packed in the app, and few existing studies in the community have systematically analyzed the behaviors of commercial app packers. In this paper, we propose PackDiff, a dynamic analysis system to inspect the fine-grained behaviors of commercial packers. By instrumenting the Android system, PackDiff records the runtime behaviors of Android apps (e.g., Linux system call invocations, Java API calls, Binder interactions, etc.), which are further processed to pinpoint the additional sensitive behaviors introduced by packers. By applying PackDiff to roughly 200 apps protected by seven commercial packers, we observe the disappointing facts of existing commercial packers. Most app packers have introduced unnecessary behaviors (e.g., accessing sensitive data), serious performance and compatibility issues, and they can even be abused to create evasive malware and repackaged apps, which contradicts with their design purposes.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; **Software security engineering**.

*Co-first author.
†Haoyu Wang is the corresponding author (haoyuwang@hust.edu.cn).

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558969

KEYWORDS

Commercial Android Packers, Dynamic Analysis, Privacy Leakage

ACM Reference Format:
Zikan Dong, Hongxuan Liu, Liu Wang, Xiapu Luo, Yao Guo, Guoai Xu, Xusheng Xiao, and Haoyu Wang. 2022. What Did You Pack in My App? A Systematic Analysis of Commercial Android Packers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558969

1 INTRODUCTION

Android apps can be easily decompiled, hacked and repackaged by adversaries. Thus repackaged apps and malware are long-lasting issues in the mobile app ecosystem. These attacks pose a serious hazard to both developers and users. To protect apps from being hacked, many security vendors have introduced app security enhancement services, i.e., *app packers*, which usually apply encryption, obfuscation and other protections to prevent the app from being analyzed by reverse engineering tools, e.g., Apktool [11] and IDA [16]. Additionally, these services adopt runtime protection to impede the dynamic analysis and unpacking of packed apps. When a packed app is launched, the packers' code will be executed before the other part of the code. In particular, the packer's code will first verify the integrity of the app and confirm the secure running environment. Then the app's code can be decrypted and executed.

To hide the technical details (i.e., protecting the packed apps from being unpacked), the app packer is usually provided to developers as a black box, i.e., in the form of an online service. Developers upload apps to the Android packing service website, and the app will be packed by the packing service in the background. After the packing is completed, developers download the packed apps and re-sign them. During the whole process, developers do not have direct access to the Android packer, except the packed app. Thus, developers have no way to know what the packing service pack in their apps and what kinds of side effects (e.g., unexpected behaviors, and performance issues) are introduced by the packing service.

Indeed, there is increasing evidence that commercial packers can introduce undesired behaviors to the original apps. For example, it was reported that Qihoo packer once embedded advertisement services in packed apps without the consent of app developers [7].

Besides unwanted side effects, app packers can be abused by attackers. In particular, packing services have been exploited to prevent malware from being analyzed, as shown in the recent works on Android packers [25, 34, 37, 38]. To study the packed malware and investigate their malicious behaviors, these works focus on how to unpack the packed apps, i.e., extracting the original `Dex` files from the packed app. Most of these works are based on dynamic analysis, i.e., unpacking the app by modifying the Dalvik runtime [10] or ART (Android Runtime) [37, 38] or relying on emulators [34]. With the development of unpacking techniques, app packers also evolve their packing services to impede these unpacking techniques. It's an arms race between packing and unpacking techniques.

To the best of our knowledge, while there exist studies on unpacking the packed apps, few existing studies in the community have systematically analyzed the behaviors of commercial app packers and their side effects. *This, however, should be more transparent to app developers, who definitely have the right to know what has been packed in their own apps.*

This Work. In this paper, we propose PackDiff, a comprehensive analysis system to inspect the fine-grained behaviors of commercial app packers. By instrumenting the Android system, PackDiff records the runtime behaviors of Android apps, including Linux system call invocations, Java API calls, Binder interactions, app component creation and thread activities. The instrumentation is done via hooking the functions in the Linux Kernel, Android Runtime and Android framework. As PackDiff works on the Android system directly, app packers cannot easily detect the presence of PackDiff during the execution of the app, and thus PackDiff can bypass the runtime protection measures (e.g., sandbox or emulator checking) of the packers. By comparing the app behaviors before and after packing, we can accurately pinpoint the additional behaviors introduced by app packers.

By applying PackDiff to 196 packaged apps protected by seven commercial Android packers (15 packer versions in total), we show that PackDiff can analyze the fine-grained behaviors of commercial Android packers (**RQ1**), including sensitive data access, network interactions, system information collection, additional created app components (e.g., services), background resident executions, and anti-dynamic analysis checks. We further show that the runtime performance and the compatibility of original apps can be affected by the commercial packers greatly (**RQ2**). Furthermore, we investigate whether the commercial packers can be easily abused by adversaries. The result is disappointing: most commercial packers can be abused to help create evasive malware and repackaged apps, which suggests that existing security vendors do not have much regulation on the proper usage of app packers (**RQ3**).

This paper makes the following major contributions:

- We proposed PackDiff, a dynamic analysis system to inspect the behaviors of Android packers comprehensively. Taking advantage of system instrumentation, PackDiff can bypass the runtime checking of packers and compare the behavior difference between the packed app and the original one. We have applied PackDiff to 196 apps protected by seven commercial app packers and their historical versions, and observe that most app packers have introduced unnecessary behaviors, including accessing sensitive data, collecting device information and network interaction, etc.
- We measured the impact of Android packers on the runtime performance and compatibility of apps. We show that the average runtime performance overhead introduced by app packers is 29% to 102% for the first start-up time and 3% to 44% for memory consumption. Worryingly, most packers would introduce serious compatibility issues.
- We revealed the dark side of app packers that they can be easily abused to help create and deliver evasive malware and repackaged apps, which contradicts with its design purpose (i.e., protecting apps from being hacked).

To boost further research, we release PackDiff [20] along with dataset used in this paper to the research community.

2 BACKGROUND

2.1 Reverse Engineering of Android Apps

Android apps are built with Java and C/C++ native code. The code is formed in the APK (Android application package) file structure which is a self-signed app. The Dalvik bytecode can be easily statically restored to source code or equivalent expression using decompilers, such as baksmali [3], dex2jar [4] and Jadx [9]. For native libraries written in C/C++ language, there are many existing mature static analysis tools for analyzing C/C++ binary files, including capstone [2] and IDA [16],

Android apps can also be dynamically analyzed in many ways. Linux provides the system call `ptrace` for dynamic tracking and debugging process running in the system. By manually debugging the app, the implementation of the app can be inspected. If only a cursory examination of the app is required, method profiling or inspection of certain function parameters and return values can be implemented through dynamic binary instrumentation techniques, such as Frida [5] and Xposed framework [6].

Therefore, taking advantage of existing app analysis techniques, Android apps can be easily decompiled, analyzed, hacked, and repackaged by adversaries.

2.2 Android App Packer

Due to the large amount of plagiarism and repackaging in the Android ecosystem [28, 31], Android app packing technique was introduced and Android packers were adopted by a large number of developers as an effective means to protect their apps. In general, the Android packers usually protect the apps from three aspects, including improving the bar of static analysis, protecting apps at runtime and preventing apps from being tampered with.

To impede static analysis, Android packers protect both `Dex` files and `so` files. The original `Dex` files of the apps are usually protected through encryption, dynamic releasing (i.e., dynamically releasing the protected data into the memory for execution during the runtime), dynamic modification (i.e., modifying `Dex` files in the memory when the app is running), obfuscation, and reimplementing with native code. Furthermore, some packers adopted virtual machine-based protection techniques, which translate Dalvik bytecode to

another customized type of bytecode and embed a customized virtual machine to interpret them when the packed app runs on a device. For so files, Android packers protect them using ELF file packer or obfuscation tools like Obfuscator-LLVM [27].

To protect apps at runtime, Android packers first examine the running environments to prevent the packed apps from running on the emulator which is leveraged by many unpacking techniques to carry out dynamic analysis, or environments with root privileges. On the other hand, Android packers will check if the app is being debugged or hooked by tools such as Frida [5] and Xposed frameworks [6]. Additionally, Android packers usually occupy the debugging interface or detect tool-specific features to protect Dex files from being dumped from memory.

To prevent apps from being tampered with, Android packers usually adopt file verification mechanism. Android packers generate a fingerprint for each code file, configuration file and resource file. Then, they calculate the file fingerprint at application startup and compare it with the original file fingerprint previously calculated. If the two fingerprints match, the app continues to execute.

Android packers are widely adopted by existing apps, especially malicious apps. Even some third-party markets require app developers to pack apps before they are published to the market [22]. Since Android packers provide a lot of protection for apps and different security vendors implement their commercial Android packers in different ways, it is difficult to analyze the detailed behaviors of packed apps directly using traditional app analysis techniques.

2.3 Android System

The Android system is a hierarchical structure that can be roughly divided into the Linux kernel, hardware abstraction layer, Android Runtime, native C/C++ libraries and Java API framework. Next, we will introduce the parts that are relevant to our work.

The foundation of the Android platform is the Linux kernel. The Android Runtime relies on the Linux kernel for underlying functionalities such as threading, file management and low-level memory management. The kernel provides functionalities for apps through system calls. Each system call is assigned a unique system call number and distributed to the corresponding handler according to the system call number in the kernel.

Android Runtime consists of the ART runtime [12] and a set of core runtime libraries. For devices running Android 5.0 or higher, each app runs in its own process and with its own instance of the ART runtime which is proposed to improve the performance of the Android system. ART runtime introduces ahead-of-time (AOT) compilation. When the app is installed, the Dex bytecode of the app is compiled to native code, which results in an ELF-format oat file. When the app is running, the generated oat file is loaded into memory, and ART runtime can find the native code corresponding to the method of any class, to execute according to the segment information stored in the oat file. In addition, Android also includes core runtime libraries that provide most of the functionalities of the Java programming language, such as the "ojluni", which stands for OpenJDK, java.lang, java.util, java.net and java.io.

Java API framework provides the entire feature-set of the Android OS through its APIs written in Java language. These APIs form the building blocks that developers need to create Android apps with rich features by simplifying the reuse of core, modular system components and services, which include content providers, view systems and many managers, such as the package manager, location manager and telephony manager.

Android system services provide interfaces for apps to call their functions. Since apps and system services run in different processes, Android provides an inter-process communication mechanism, i.e., Binder, for apps to interact with both Java system services and native system services.

Table 1: An overview of system instrumentation

Collected Information	Modified System Level	Behavior
Java API call	Android runtime	Sensitive data access
		Anti-dynamic analysis check
Binder interaction	Android framework	Sensitive data access
Linux system call invocation	The Linux kernel	Internet interaction
		System information collection
		Anti-dynamic analysis check
Component creation	Android framework	Additional component creation
Thread activity	Android framework	Background resident execution

3 OVERVIEW OF PACKDIFF

3.1 The Need of System Instrumentation

To perform a systematic investigation of Android packers, we focus on six types of behaviors related to packers, including 1) sensitive data access, 2) network interaction, 3) system information collection, 4) app component creation, 5) background resident execution and 6) anti-dynamic analysis check. These are all possible behaviors we could enumerate to characterize the packers.

However, since commercial Android packers provide many protection measures to prevent apps from being analyzed, it is difficult to investigate the behaviors of packed apps directly using existing static or dynamic techniques [5, 6, 11, 16]. To bypass the protection measures, we take advantage of system instrumentation to monitor the runtime behaviors of packed apps. System instrumentation technique refers to directly inserting our customized monitoring code into the system, which allows us to monitor app behaviors and is difficult for packers to detect.

Thus, we need to select the proper instrumentation points to get a comprehensive analysis of the packers. First, the app obtains sensitive data through specific Java APIs which need to interact with system services through Binder and retrieve sensitive data. Second, unlike sensitive data, some system information that is not relevant to the user is stored in certain system files, and apps can obtain system information by accessing these files. Third, apps usually transfer data externally via files or networks, and there are many ways to access files and networks either through Java or native code. But whichever method is adopted, it will eventually be processed by the kernel through system calls. Fourth, for additional component creation and background resident execution, apps also need to call related Java APIs. At last, security vendors can provide different anti-dynamic analysis mechanisms, and most of them need to rely on specific APIs or system calls.

Therefore, through the recording of *Java API invocation*, *Binder interaction*, *Linux system call invocation*, *component creation* and *thread activity*, we can achieve the aforementioned analysis.

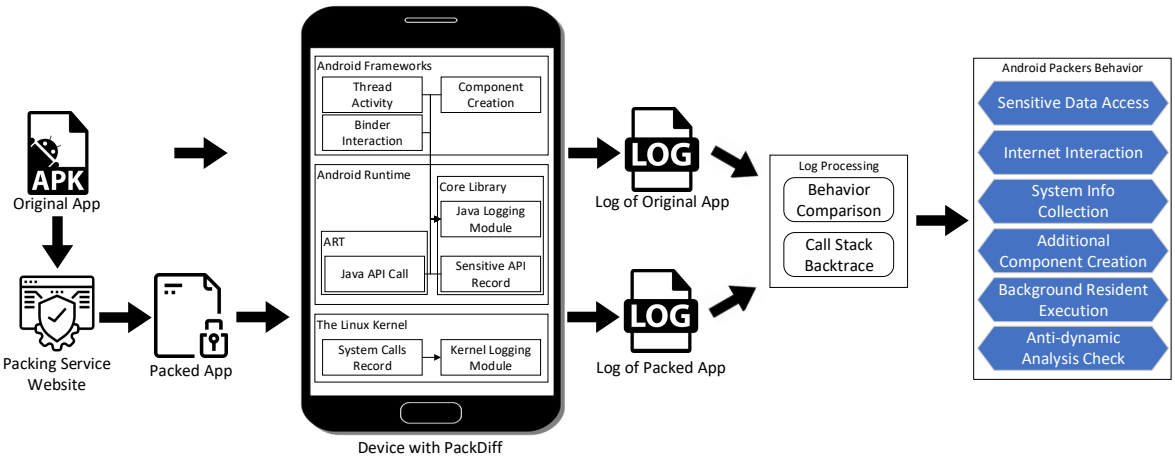

Figure 1: The overall architecture of PackDiff.

3.2 Overview

The overall architecture of PackDiff is shown in Figure 1. We instrument the Linux kernel, Android runtime and Android frameworks to collect information on Linux system call invocation, Java API call, Binder interaction, component creation and thread activity. We monitor the detailed runtime behaviors of the packed app and the original one using PackDiff, and the generated log information is processed for comparison in the log processing module. In this way, we can accurately pinpoint the *additional behaviors* introduced by packers. Table 1 summarizes our modifications to the Android system, the collected information and the corresponding behaviors we can infer.

4 THE DETAILS OF PACKDIFF

In this section, we depict each part of PackDiff in detail.

4.1 Linux Kernel Instrumentation

We aim to record all necessary system calls invoked by the app by instrumenting the Linux kernel. To this end, we first need to identify the threads of the target app and only record the system calls of the corresponding threads. When a system call is invoked by the target app, PackDiff records the system call name, parameters and return values. The parameters and return values can be leveraged to capture the behaviors of the apps. In addition, we record the call stack of each system call to distinguish the behaviors of Android packers. During log processing, we use system call records to maintain the file descriptor table and memory mapping, supplement the call stack and file-related behaviors of apps.

4.1.1 Flagging Target Threads. As many apps are running simultaneously in Android system, we flag the threads of the target app in the Linux kernel to reduce performance overhead when it starts. Our system instrumentation should only be activated for the flagged threads. For this purpose, we customize a tag for the target threads and obtain their records of system calls. In the Linux kernel, each thread is represented as a `task_struct` structure, which provides a member variable `flags` to tag the threads. This helps us identify all threads of the target app by specifying the customized tag. After the target app's main thread is started, our customized tag would be set in the `task_struct` structure through system call `prctl`, which can manipulate various aspects of the behavior of the calling thread and determine the action to be performed based on its `option` parameter. Specifically, we add a branch supported by the `option` parameter, and use this branch to flag the main thread of the target app and start tracing. After that, each time a thread is created by system calls (e.g., `fork`), the entire `task_struct` structure is copied and inherits our customized tag. As a result, all child threads are flagged and traced by PackDiff.

Table 2: The tag of parameters and return values

Type	Description	Resolve at Runtime	Resolve in Log Processing
FD	Identifier for a file		√
PID	Process identifier		√
ADDR	Memory mapped address		√
Path	File path string	√	√
Data	Structure parameters	√	√
ARGV	Parameter list of execve	√	√

4.1.2 Recording System Call Invocation. To record the invocation of system calls, we leverage the debugging interface for system calls provided by the kernel. The kernel relies on `_TIF_SYSCALL_WORK` to determine which threads need to be debugged. When a thread's tag matches `_TIF_SYSCALL_WORK`, Linux kernel will call function `syscall_trace_enter` and `syscall_trace_exit` before and after the execution of system calls, respectively, and we can access the information of the invoked system calls in both functions. Thus, we set our customized tag to `_TIF_SYSCALL_WORK` to activate the functions in the target thread and modify the functions to record the invocation of system calls.

4.1.3 Resolving System Call Parameters. The parameters of some system calls can provide useful information regarding the behaviors of packed apps. Thus, we seek to resolve these parameters, e.g., mapping the file descriptor to the opened file name. We mainly retrieve six types of parameters and return values, as shown in Table 2. For the consideration of system performance, we put most of the resolution in the log processing phase. Particularly, for some parameters and return values of pointer type (i.e., Path, Data and ARGV), it is necessary to parse the content at runtime.

For parameters of type FD (i.e., an identifier for a file), we need to know the state of the file description table which records the mapping between file descriptors and opened file names, when

each system call is invoked. To eliminate potential performance overhead, PackDiff records the initial state of the file descriptor table at runtime and further maintains the table information in the log processing phase after the test is completed. To be more specific, if a system call is found to create a file descriptor, we add it to the maintained file descriptor table; if the invoked system call is used to release a file descriptor, we delete the corresponding item in the table. By maintaining the file description table, we can know every file that the app touches.

For parameters of type ADDR (i.e., memory mapped address), memory mapping maintenance is completed similar to file descriptor maintenance, in which the initial state is first recorded and we continue to maintain during the log processing based on record of system call `mmap` and `munmap`.

4.1.4 Log Call Stack. After retrieving the above system calls, PackDiff uses the system call stack backtrace to determine whether a system call is invoked by the packer. Using the address in each call stack and maintained memory mapping, the invoker of the system call can be obtained. For invocations of system calls that exist only in the record of packed apps, we consider them to be behaviors from the packer. To perform call stack backtrace in ARM architecture, the frame pointer needs to be activated during Android building phase. Figure 2 illustrates the structure of the call stack in 32bit ARM architecture after the frame pointer is active. R7 register stores the frame pointer, which points to the frame pointer of the last stack, and the return address is stored in `lr` register.

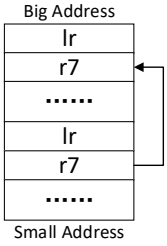

Figure 2: The structure of call stack in 32bit ARM

During the system call stack backtrace, we determine if the backtrace should be ended by checking whether the frame pointer is still in the stack and whether the address pointed to by the frame pointer is increasing.

4.1.5 Logging Module in the Linux Kernel. We next introduce the logging module which records the logs from the kernel. In the Linux kernel, the log messages are generally output by the function `printk`. However, function `printk` does not support the output of large amounts of logs. First, the max buffer size is limited to 2 MB, and anything over 2 MB will overwrite the previous logs. Second, when logs are output at a high frequency in a short period, part of the logs will be discarded. Therefore, PackDiff implements an additional logging module in the Linux kernel. As the size of logs grows, the cache size is dynamically extended, and a new file where the logs are output is added to the `proc` filesystem.

4.2 Android Runtime Instrumentation

To record Java APIs called by the app, we set instrumentation in the ART runtime and core runtime libraries in Android Runtime.

4.2.1 ART Runtime. We instrument the entry point of the Java API calls in ART runtime to record all Java APIs called by the app. The ART runtime has two execution modes, one using the interpreter to execute the Java code, and the other executing native code generated by ahead-of-time compilation. There are two entry points when ART runtime executes a Java method. When an app calls a compiled Java method, ART runtime executes the method through method `Artmethod::Invoke` in `art_method.cc`. When an app calls a method in Java code, ART runtime interprets the method through function `DoCall` in `interpreter_common.cc`. Based on the instrumentation, we can identify API calls related to sensitive data access and anti-dynamic analysis.

4.2.2 Core Libraries. To monitor some specific behaviors of the app (e.g., encryption and system command operation) in detail, we set instrumentation in related runtime libraries. Android includes a set of core runtime libraries that provide most of the functionalities that the Java API frameworks use, including functions for encryption and system command operation. We record the corresponding data in these functions (e.g., invoked system commands).

Additionally, we implement a logging module within package `java.util` to output information from core libraries and Android frameworks, which enables code in core libraries and Android frameworks to directly import the class of logging system.

4.3 Android Framework Instrumentation

We next instrument the Android frameworks to collect information on Binder interaction, thread activity and component creation.

4.3.1 Binder. Since apps and system services run in separate processes, Android provides *Binder*, an inter-process communication mechanism, for apps to interact with system services. As shown in Figure 3, Android frameworks use native library `libbinder` to access Binder , and `libbinder` invokes `ioctl` to access the Linux Kernel through `/dev/binder`. Finally, the Binder driver in the Linux Kernel completes the interaction between apps and system services.

To record the behaviors of the app using Binder, we instrument the library `libbinder`. We modify the method `Bpbinder::transact`, in which we can record the accessed system service, called system function and the corresponding parameters.

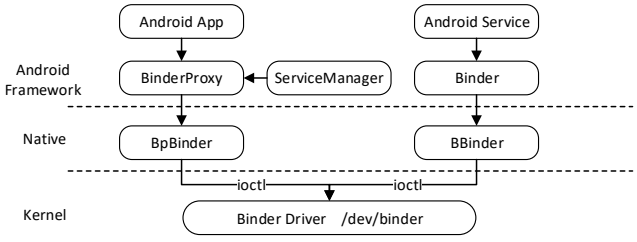

Figure 3: The Binder architecture in Android

4.3.2 Thread. To analyze the thread activities and the background resident execution behavior of apps, PackDiff monitors the creation and completion of threads. We record thread creation via the instrumentation of `Thread` class. Android allows an app to start a thread through a new instance of the Java class `Thread` and calling method `Thread.start`. We record the ownership of new threads in method `Thread.start`. The created thread will execute the code

in method `Thread.run` after it starts. In one case, the caller overrides method `Thread.run`. In the other case, a `Runnable` object is passed to Thread constructors by the caller. So we record the actual type of the class `Thread` in runtime and the type of the parameter `Runnable` before the execution of the new thread starts.

To monitor the running threads, we add a timer in Android frameworks, which starts after the app is launched, and invoke the method `Thread.getAllStackTraces` every 10 seconds. When a thread disappears from the running thread list, we consider the thread finished. Furthermore, we can figure out the threads which reside in the background for a long time.

4.3.3 Android App Components. PackDiff also monitors the behaviors related to app components. Android provides four different types of app components, i.e., activity, service, broadcast receiver and content provider, which serve distinct purposes. Three components in Android must be statically registered in the file `AndroidManifest.xml`, while the broadcast receiver can be registered dynamically through the method `Context.registerReceiver`. Thus, we instrument this method to capture the dynamic registration of broadcast receiver.

4.3.4 App Launch Process. To start PackDiff's monitoring of the target app, we activate our instrumentation during the app launch process which can be roughly divided into the following steps:

Step-1 The `Zygote` process creates an app process by executing fork through function `forkAndSpecialize`.
Step-2 The new process initializes its environment and ART runtime, and then executes the method `ActivityThread.main`.
Step-3 `ActivityThread` object performs Java-related initialization and sends a message `BIND_APPLICATION` to the message queue by calling method `bindApplication`.
Step-4 `ActivityThread` object processes the message through method `handleBindApplication` and completes the launch process.

We determine whether the starting app is the target app according to the package name information in `forkAndSpecialize` and `handleBindApplication` function. We turn on the record switch in the Linux Kernel through system call `prctl` after zygote executes `fork`, and turn on the record switch for Android runtime and Android framework in method `handleBindApplication`.

5 EVALUATION SETUP

5.1 Dataset

For a systematic evaluation of Android packers, we select seven real world commercial Android packing services currently available for individual developers, including Baidu [13], Bangcle [14], Ijiami [17], Qihoo [21], Naga [19], Manxi [18] and Tencent [23]. These are popular app packing services that serve millions of app developers. As aforementioned, these app packers are black boxes, and they can always be updated to deal with increasingly powerful unpacking techniques over time. To investigate whether the behaviors of these commercial Android packers will also change over time, we collect apps that are packed by historical versions of commercial packing services from previous research [34, 36]. Besides, to examine the service's capability of inspecting the uploaded app, we prepare some repackaged apps and malicious apps for evaluation.

Table 3: Summary of our dataset.

Android Packer	Dataset2017	Dataset2020	Dataset2022	DatasetMix
Baidu_2017	10			
Qihoo_2017	10			
Baidu_2020		10		
Bangcle_2020		10		
Ijiami_2020		10		
Qihoo_2020		10		
Tencent_2020		10		
Baidu_2022			20	
Bangcle_2022			20	
Ijiami_2022			20	
Manxi_2022			20	
Naga_2022			2	
Qihoo_2022			20	
Qihoo#_2022			4	
Tencent_2022			20	
Malicious app				15
Fake app				15
Total (Packed/Original)	20/10	50/10	126/20	30

Table 3 summarizes the datasets used in our experiments. In total, we have compiled a list of 246 apps, including 196 packed apps, 20 original apps, and 30 apps used for evaluating whether these commercial services can be abused by adversaries.

5.1.1 Dataset2017. We collect 10 apps from [34], each of which was packed by two commercial packing services, i.e., Baidu and Qihoo, in April 2017. Thus DataSet2017 has a total of 20 packed apps protected by historical versions of packers, and 10 original apps. It is worth noting that these 10 apps declare different permissions, which will have a great impact on the behaviors of the packers, and we further discuss this in detail in Section 6.

5.1.2 Dataset2020. We collect the same 10 apps as Dataset2017 from [36], each of which was packed by five commercial packing services, including Baidu, Bangcle, Ijiami, Qihoo and Tencent, in January 2020. Thus Dataset2020 has a total of 50 historical versions of packed apps corresponding to the 10 original apps.

5.1.3 Dataset2022. Dataset2022 is created by interacting with the latest packing services in April 2022, including all the seven accessible packing services aforementioned. On the basis of the same 10 original apps as the first two sets, we collect another 10 open-source apps from F-Droid [15], to expand the dataset. Then, we seek to pack these apps using each of the packing services. As a result, we successfully packed all the apps using six of them, except for Naga where only 2 apps were successfully packed since it allowed us to pack self-developed apps only. Besides, we notice that Qihoo supports additional user-selected packing options (e.g., x86 platform support, signature verification, etc.), we thus tried to check all the options to obtain an alternative version of the packing service (marked as Qihoo#). However, some packing options require specific permissions declared by the app to take effect, which prevents many apps from being packed in this way. At last, we have 4 apps successfully packed in this way (i.e., by Qihoo#). As a result, Dataset2022 consists of 126 versions of packed apps (from 20 original apps) in total using current packing services.

To distinguish between different versions of the packing services for the same app, we mark the apps in Dataet2017, Dataset2020 and Dataset2022 in the form of "packing service name_packing time", like "Qihoo_2017", "Baidu_2020", "Manxi_2022".

5.1.4 DatasetMix. To explore whether the commercial app packing services can be abused by adversaries, we prepare a set of malicious apps and fake apps that counterfeit well-known ones (e.g., fake Facebook and WeChat apps), and intend to upload them to the packing services and check whether these commercial services have strict regulations on the uploaded apps, e.g., malware checking or developer identify checking. Specifically, we collected 15 malicious apps from 5 malware families in the most widely used MalGenome dataset [39]. We collected 15 well-known apps from Google Play and Baidu App markets, which were then resigned by us to create some fake apps.

5.2 Research Questions

For a systematic summary of the commercial packing service, we investigate three main research questions:

- RQ1 What behaviors will the packing services add to the app?
- RQ2 What side effects (e.g., performance and compatibility issues) will the Android packers bring to the app?
- RQ3 Do the packing services strictly inspect the app before they pack the app?

To answer RQ1, we leverage PackDiff to analyze the behaviors of packed apps in Dataset2017, Dataset2020 and Dataset2022, and compare them with the behaviors of original apps to pinpoint the difference. To answer RQ2, we test the impact of Android packers on app performance through the comparison of the performance difference between the original apps and the packed apps. For compatibility comparison, we purchased commercial compatibility tests on various kinds of real-world Android devices to aid our study. To answer RQ3, We upload the malicious apps and fake apps in DataseMix to the packing service website, and check whether we can successfully pack them.

5.3 Experimental Environment

We conduct the experiments mainly on Android 9.0.0 with PackDiff and the Linux kernel version in Android is 3.18.100. Particularly, due to the compatibility issue of apps in Dataset2017, we test these apps on Android 7.1.0 with PackDiff. In addition, to maintain the consistency of the experimental environment, all app performance tests were conducted in the unmodified Android 7.1.0 on the same smartphone. All the experiments are carried out on Google Pixel.

6 EVALUATION RESULTS

6.1 RQ1: Inserted Behaviors

Methodology. We apply PackDiff to test every packed app in Dataset2017, Dataset2020 and Dataset2022, as well as their original ones before packing, to make pairwise comparisons. The testing process goes through five steps: 1) reboot the device and reset the experimental environment; 2) install the app and grant all app requested permissions; 3) activate monitoring of apps and start to collect logs; 4) launch the app and perform the automated clicks using automation testing tool `uiautomator2` [8]; 5) after the automated test has lasted five minutes, we stop the testing. Finally, we process the logs to analyze and compare the behaviors of each app pair. Note that, for each app pair, we manually devise a testing script to trigger the main functionalities of the app, which is used in step 4. We next present our experimental results in terms of six types of behaviors added by app packers.

6.1.1 Sensitive Data Access. As shown in Table 4, we observe that some commercial packers show sensitive behaviors without consent from app developers, including declaring new permissions, dynamically checking permissions and calling sensitive APIs.

Declaring Additional Permissions. Android packers that provide protections for apps such as encryption and verification, should be able to complete without using any additional permissions. However, Baidu_2022 and Baidu_2020 add new permission declarations to `AndroidManifest.xml`, including ACCESS_NETWORK_STATE, INTERNET, READ_PHONE_STATE, ACCESS_WIFI_STATE, RECEIVE_BOOT_COMPLETED, GET_TASKS and READ_EXTERNAL_STORAGE. With these permissions, Android packers can obtain the user's IMEI, phone number, SIM card information, IP address and MAC address, etc., and leak sensitive data through the Internet. These permissions are not necessary for the functionalities of packers, which should be more transparent to developers. Note that, Qihoo packer explicitly declares that it needs apps with permissions including ACCESS_NETWORK_STATE, READ_PHONE_STATE and INTERNET, to make the additional function `full-time crash monitoring` work. The permissions should be declared by developers, thus we do not regard it as additional permission declaring.

Dynamically Checking Permissions. Besides declaring new permissions, some app packers can also make use of the permissions requested by the app to collect sensitive data stealthily. In general, the app packers first check whether the app has a certain granted permission at runtime. Our experimental results show that Baidu_2020, Baidu_2022 and Qihoo_2017 performed dynamic permission checks through related APIs. It is worth noting that Qihoo_2017 does not add any new permission declarations, but it checks for permissions when the app runs. If the app requests some permissions and users grant them at runtime, Qihoo_2017 uses these permissions to perform the sensitive behaviors.

Calling Sensitive APIs. After the packed app has been granted sensitive permissions, packers can use the permissions to access the corresponding sensitive data. Our experimental results show that Baidu_2020, Baidu_2022, Qihoo_2017, Qihoo#_2022 and Tencent_2020 have additional sensitive API calling behaviors compared with the original apps. Such sensitive behaviors include obtaining IMEI, phone number and SIM card information, obtaining network information, obtaining Wifi information, and obtaining location information, which are not directly related to their main functionalities. We further analyze the behaviors of the packers in the absence of user interactions and demonstrate that there are no specific trigger conditions for these sensitive behaviors.

6.1.2 Internet Interaction. We observe that some packers perform additional network interaction behaviors. We query the destination IP address of their network connection from the log, and find that Baidu_2020, Baidu_2022, Qihoo_2017, Qihoo#_2022 and Tencent_2020 will frequently communicate with the server of the packing service provider. For example, through analyzing the behavior logs of encryption-related APIs, we find that Baidu packer encrypts and encodes the information including the device model, system version, app version and device MAC address, and then transmits it to the server belonging to Baidu.

Table 4: Summary of sensitive data access behavior of commercial Android packers

Android Packer	Declaring New Permissions	Dynamically Checking Permissions						Calling Sensitive APIs			
		NETWORK	WIFI	BLUETOOTH	INTERNET	AUDIO_SETTINGS	PHONE	Telephony	Wifi	Connectivity	Location
Baidu_2020	√		√				√	√	√	√	
Baidu_2022	√			√		√		√			
Qihoo_2017		√	√		√		√	√		√	√
Qihoo#_2022								√	√		
Tencent_2020								√		√	

Table 5: System information accessed by Android packers

File Path	File Content	Qihoo_2020	Qihoo_2022	Qihoo#_2022	Baidu_2022	Tencent_2020
/proc/cpuinfo	Device CPU Info				√	
/proc/meminfo	Device Memory Info				√	
/proc/version	Linux Kernel Compile Info			√		
/proc/net/tcp	TCP Connection Info	√				
/system/build.prop	Android Compile Info	√	√	√		√

6.1.3 System Information Collection. Android packers can obtain Android device information by accessing system files, which contain basic system information such as CPU and memory information, Android version, Linux kernel version, etc. Table 5 shows the system files accessed by Android packers, which are mainly used to check whether the app is running on the emulator. Some system information can also be obtained through external processes. For example, Ijiami_2020 and Tencent_2020 get system properties through command `getprop` while Baidu check whether SELinux is enforced by command `getenforce`.

6.1.4 App Component Creation. We observe that some packers created new app components, as listed in Table 6. By inspecting the `AndroidManifest.xml`, we find that Baidu_2017, Baidu_2020, Qihoo_2017 and Bangcle_2022 add new components to apps in the file. Further, our instrumentation in method `Context.registerReceiver` revealed that Baidu_2022, Qihoo_2022, Qihoo#_2022 and Tencent_2020 create broadcast receiver dynamically and listen for system event CONNECTIVITY_CHANGE, and Baidu_2020 registers broadcast receiver for system event SCREEN_ON, SCREEN_OFF and PACKAGE_REMOVED. These additional app components are supposed to serve the packing functionality, but our system monitoring suggests that some of them are not quite related to the packing function. Taking broadcast receiver as an example, monitoring system events like CONNECTIVITY_CHANGE and SCREEN_ON, is obviously not a necessary behavior for app packing.

Table 6: Android components added by Android packers

	Activity	Service	Content Provider	Broadcast Receiver
Baidu_2017			√	
Baidu_2020	√	√	√	√
Baidu_2022				√
Bangcle_2022			√	
Qihoo_2017	√	√	√	√
Qihoo_2022				√
Qihoo#_2022				√
Tencent_2020				√

6.1.5 Background Resident Execution. Some Android packers show background resident execution behavior. By tracing the thread operation, we find that Baidu_2022, Baidu_2020 and Tencent_2020 create additional threads and keep running in the background after the app is launched. These packers interact with their servers in separate threads. In addition, Bangcle_2022 and Tencent_2020 also create new threads where they detect if some important files (e.g., the app's internal storage directory and ANR (Application Not Responding) trace file directory) are modified during app running.

6.1.6 Anti-dynamic Analysis Checking. To prevent the app from being dynamically analyzed, the packers perform a set of anti-dynamic analysis checking to protect the app, and the results are shown in Table 7. PackDiff can detect the security checks that depend on the access to key functions or files, including:

1) Detecting Java debugging through Java API `Debug.isDebuggerConnected`.
2) Detecting whether the key file has been read, modified or created via the system call `inotify`.
3) Detecting manual debugging by calculating execution time using system call `clock_gettime`.
4) Detecting `ptrace` debugging by checking the tracePid field in `/proc/PID/status`.
5) Determining if the app is running in an emulator through checking for the existence of virtual machine-specific files, like `/dev/qemu_pipe`.
6) Determining if the app is running in the device with root privilege through checking for the existence of feature files, like `/system/bin/su`.

Table 7: Anti-dynamic analysis methods used by packers

	Detect JDB Debugger	Calculate Execution Time	Protect File	Detecting Emulator	Detecting ptrace	Detecting Root
Baidu_2017	√					
Baidu_2020	√					
Baidu_2022	√					
Bangcle_2020			√			
Bangcle_2022					√	
Ijiami_2020		√		√	√	
Ijiami_2022	√				√	
Manxi_2022	√		√	√	√	
Naga_2022	√			√	√	
Qihoo_2022	√				√	√
Qihoo#_2022	√				√	√
Tencent_2020			√			√

6.1.7 The Evolution of Android Packers Behaviors. By comparing the behaviors of Android packers with multiple versions (i.e., versions 2017, 2020 and 2022), we can see that the inserted irrelevant behaviors from the packing services are gradually decreasing. Among the latest packing services, only Baidu still accesses sensitive data without the developer's knowledge.

Answer to RQ1: *Experimental results show that PackDiff can effectively analyze the behaviors of packed apps. We observe that some commercial packers would involve behaviors that are unnecessary to their main functionalities, e.g., collecting sensitive information. These behaviors, however, should be more transparent to developers who use these services to pack their own apps.*

6.2 RQ2: Side Effects

Methodology. We analyze the side effects introduced by Android packers by comparing each app pair (i.e., the packed app and the original one), in terms of performance and compatibility. For each tested app, we start and stop the app 5 times continuously after installation on the same device with unmodified system[1], and record the first start-up time, average start-up time and memory consumption through `ActivityManager` and `dumpsys` tools.

In addition, we purchased an app compatibility testing service with various kinds of real Android devices to evaluate app compatibility. We choose WeTest [24], which is a popular app testing platform that helps developers identify compatibility issues such as installation failure, crashes, and app not responding issues using dynamic analysis on real devices. It costs about $360 for 1,000 compatibility tests (i.e., testing one app on one specific device is a test). Note that, due to financial constraints, we were not able to test all the packed apps on all kinds of devices. Thus, we selected 100 apps from our dataset, which contain all packing service versions we considered, and the testing devices covered the major smartphone manufacturers and all system versions from the minimum supported Android version to Android 12. This comprehensive testing enables us to study the compatibility issues introduced by packers.

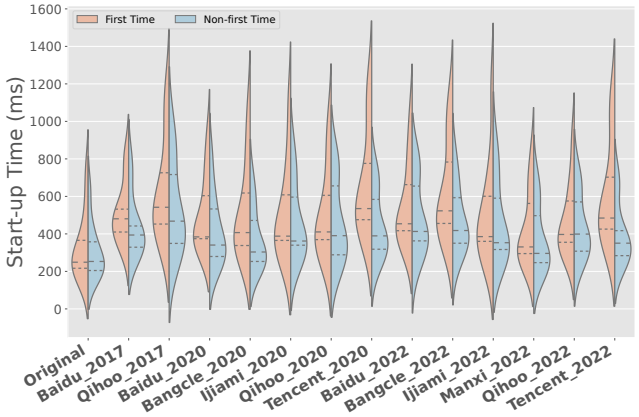

Figure 4: An analysis of the Start-up time.

6.2.1 The Impact on App Performance. We analyze the impact of packers on app performance from two aspects: *start-up time* and *memory consumption*. Since packers need to perform protection measures when app starts, it inevitably takes longer time for start-up than the original app, and consumes more hardware resources.

Start-up Time. Figure 4 shows the *average start-up time* and *first start-up time* of packed apps and original apps (the first column). App packers can introduce significant performance overhead. The overhead of *average start-up time* ranges from 22% to 76%, and the overhead of *first start-up time* ranges from 22% to 102%. Among all the packing services, Qihoo_2017, Ijiami_2020 and Ijiami_2022 are the ones that introduce the largest startup time overhead. More specifically, the average start-up time of original apps is 303ms, while the average start-up time of Qihoo_2017, Ijiami_2020 and Ijiami_2022 are 532ms, 461ms and 458ms respectively. Qihoo_2017 performs a lot of file operations during the app startup, and Ijiami performs a lot of string manipulation and file operations, which may be the reason for the slow start of the packed apps. It is worth noting that for all packing services except Ijiami_2020 and Ijiami_2022, the non-first start-up time is significantly shorter than the first start-up time. The reasons might be that there are some one-time operations, and they further perform caching operations for speeding up.

Memory Consumption. The average memory consumption overhead introduced by packers ranges from 3% to 44%. Qihoo_2017 (38%) and Baidu_2020 (44%) introduced the largest overhead while Tencent showed the least impact on memory usage (3%).

[1]The unmodified system refers to the original Android system without PackDiff, in order to prevent our modifications to the system from compromising performance of the app.

Table 8: Compatibility issues caused by Android packers

Packer Version	5	6	7	8	9	10	11	12
Baidu_2017				●	●	●	●	●
Qihoo_2017				●	●	●	●	●
Ijiami_2020							●	●
Bangcle_2020							○	○
Tencent_2020							●	●
Baidu_2022	△	△	△	△	△	△	△	△
Naga_2022					□			
Manxi_2022							○	

Notes : ● stands for All packed apps crash, ○ stands for part of packed apps crash, □ stands for part of packed apps are not responding, △ stands for part of packed apps fail to install.

6.2.2 The Impact on App Compatibility. To our surprise, most of app packers would introduce compatibility issues, as shown in Table 8. It is worth noting that all original apps were able to pass the compatibility tests. We can understand that historical versions of packing services are prone to have compatibility issues, as they did not consider the newer Android versions by design. However, we observe that the latest versions of Baidu, Naga and Manxi packers, can also cause serious compatibility issues. For example, an app (mobi.maptrek) cannot be installed on any Android versions after it was packed by Baidu, a self-developed app (com.rampage.complex) packed by Naga have no responding issue that we can not access its main UI, and a game app (app.crossword.yourealwaysbe.forkyz) packed by Manxi has been found crash issues on Android 11.

Answer to RQ2: *Unfortunately, most of the commercial packers would introduce serious performance overhead and compatibility issues to the original apps. Even the latest versions of app packers can cause issues including failing to install, app crash, etc. App developers should pay special attention to such issues when they are using these commercial packers. Sadly, these issues might be hard for developers to fix.*

6.3 RQ3: Abusing of App Packers

Methodology. As aforementioned, we aim to investigate whether app packers can be used to facilitate the creation of *evasive* malware and repackaged apps, i.e., whether they will check the submitted apps. Note that, we only test five commercial packers in this experiment, due to the limitation of Naga and Manxi packers, i.e., strict manual inspection and restricted usage times, thus we did not include Naga and Manxi in this experiment.

Result. Disappointingly, we find that most services do not have strict inspections on user-uploaded apps, which helps the spread of repackaged apps and malicious apps.

Table 9: Evaluation results of packing malware and fake apps

	Baidu	Qihoo	Ijiami	Tencent	Bangcle
Malware	●		●		○
Fake app	●	●	○	○	●

Notes : ● all apps can be packed, ○ part of apps can be packed.

For the 15 selected malware samples, all of them can be successfully packed using Baidu and Ijiami packers without showing any warning messages. As a comparison, Qihoo and Tencent can detect the presence of security risks in malware and refuse to pack all the malware samples. As to Bangcle, we can successfully pack 13 malware samples. It suggests that Baidu and Ijiami packing services did not perform any security checks on the submitted apps, while Qihoo and Tencent have the strongest security checking ability.

As to the fake apps resigned from well-known ones, Baidu, Qihoo and Bangcle did not show any warning and the apps can be successfully packed. Ijiami and Tencent can identify a part of them and refuse to pack the apps with messages like "The app is not developed by the user". It suggests that Ijiami and Tencent may have maintained a database of well-known apps and their corresponding developer signatures, and they will check the authorship of the submitted "well-known" apps.

Answer to RQ3: *Our exploration suggests that these commercial packing services can be indeed abused by adversaries to create evasive malware or repackaged apps.*

7 DISCUSSION

Implications. All the seven commercial packers we studied in this paper are widely used services adopted by millions of apps. However, we have revealed the disappointing facts that they would introduce serious issues regarding user privacy, app performance and compatibility issues, and even can boost the creation of malware. All of these commercial packers are designed by security vendors, we argue that they should pay more attention to issues about user privacy, service side effects, and service regulation. Further, for the unsuspecting app developers who have adopted app packers in their apps, they should pay special attention to the potential side effects the packers introduced, and seek measures to eliminate the impacts. Third, for security researchers, we believe there is much room for analyzing and optimizing existing mobile app services including app packers and other app enhancement services. As a result, we will release all the artifacts used in this paper to the research community.

Limitations. Despite the encouraging contributions, this work has three limitations. First, PackDiff can only be deployed on the AOSP system and Google supported devices. Through system instrumentation, we can easily bypass the detection provided by Android packers and record the information we need. However, implementation based on the system instrumentation can also be undesirable. For example, We cannot make modifications to customized Android OS without open source code. Thus we cannot deploy PackDiff to devices that are not supported by AOSP. Second, PackDiff can only record the app's invocation behaviors, but cannot analyze the execution flow inside the function. For example, after the Android packers obtain the private data, the transfer process of the private data within the process cannot be recorded by PackDiff. Third, due to the limited financial budget, we only focus on free versions of the commercial packing services. We will try to analyze the behaviors of the paid versions of the packing services in the future.

8 RELATED WORK

Android Unpacking. Android unpacking techniques have attracted the attention of many researchers[25, 33]. If Android packers can be cracked and the `Dex` files are extracted from the packed app, the original app can be further analyzed using existing static analysis tools. Kisskiss [1] attached the app process using `ptrace`, and searched for `Dex` files or `ODex` file header tags in the memory of the packed app process. Then Kisskiss dumped the `Dex` bytecode from memory. Dexhunter [38] modified both ART and Dalvik virtual machine, read the temp path of `Dex` files from the virtual machine through method hook when the packers call the corresponding APIs to load the decrypted `Dex` code into the virtual machine. Similar to us, DroidUnpack [25] is an Android packing analysis framework based on a whole-system emulation. DroidUnpack mainly focuses on packing-related behaviors, while our work focuses more on the unnecessary behaviors introduced by Android packers and their side effects. In addition, DroidUnpack relies on emulators, which are easily detected. PackDiff is an on-device system and is more invisible to packers.

System Modifications. There are many studies [30, 35–37] that adopt a system modification approach. Modifying the system allows researchers to operate at the low level of the system. TaintDroid [26] implemented a dynamic taint analysis framework by modifying the data structure of the Dalvik virtual machine and adding its taint markers. The flow of private data can be tracked at multi-levels, and thus behaviors that leak private data can be found. VPDroid [29] implemented a transparent Android OS level virtualization platform tailored for security testing based on a customized Android system. It allowed security analysts to customize different device artifacts in a virtual phone without user-level API hooking. Toller [32] implemented a tool to provide efficient infrastructure support for UI Hierarchy Capturing and UI Event Execution to Android UI testing tools based on modifications of Android frameworks.

9 CONCLUSION

In this paper, we perform a systematic study of commercial Android packers. By instrumenting Android system, we implement PackDiff, a dynamic monitoring framework for app packer analysis. We design a series of experiments to investigate the behaviors and side-effects introduced by app packers, and reveal a number of disappointing facts. We advocate the community to pay more attention to app packers, and the Android packing services should insert as few unnecessary behaviors as possible to protect users' privacy, reduce the negative impact on apps, and enforce regulations on the proper usage of app packers.

ACKNOWLEDGEMENTS

This work was supported in part by the National Natural Science Foundation of China (grants No.62072046, No.61873069), the Fundamental Research Funds for the Central Universities (HUST 3004129109), and Hong Kong RGC Projects (No. PolyU15219319, PolyU15222320, PolyU15224121)

REFERENCES

[1] 2014. Kisskiss: Android Unpacker presented at Defcon 22: Android Hacker Protection Level 0. https://github.com/strazzere/android-unpacker.

[2] 2014. The Ultimate Disassembly Framework – Capstone. https://www.capstone-engine.org/.

[3] 2015. baksmali: an disassembler for the dex format used by dalvik. https://github.com/JesusFreke/smali.

[4] 2015. dex2jar: Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar.

[5] 2016. Frida: A world-class dynamic instrumentation framework. https://frida.re/.

[6] 2016. Xposed: a framework for modules that can change the behavior of the system and apps without touching any APKs. https://repo.xposed.info/module/de.robv.android.xposed.installer.

[7] 2017. Qihoo packing service embeds charging advertisements for third-party applications. https://www.zhihu.com/question/55519031?sort=created.

[8] 2018. uiautomator2 - A library provided by Google for Android automated testing. https://github.com/openatx/uiautomator2.

[9] 2019. Jadx: Dex to Java decompiler. https://github.com/skylot/jadx.

[10] 2020. Android Runtime (ART) and Dalvik. https://source.android.com/devices/tech/dalvik.

[11] 2022. ApkTool:A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.

[12] 2022. ART runtime. https://source.android.com/docs/core/dalvik.

[13] 2022. Baidu Inc. https://app.baidu.com.

[14] 2022. Bangcle Inc. https://www.bangcle.com/.

[15] 2022. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/.

[16] 2022. IDA:State-of-the-art binary code analysis tools. https://hex-rays.com/.

[17] 2022. Ijiami Inc. http://www.ijiami.cn/.

[18] 2022. Manxi Inc. https://www.manxi-inc.com/.

[19] 2022. NAGA IN Inc. http://www.nagain.com/.

[20] 2022. PackDiff. https://github.com/PackDiff/PackDiff.

[21] 2022. Qihoo360 Inc. https://dev.360.cn/.

[22] 2022. Summary of App Upload App Market Issues. https://wenku.baidu.com/view/4bc04063cb50ad02de80d4d8d15abe23482f03db.html.

[23] 2022. Tencent Inc. https://cloud.tencent.com/.

[24] 2022. WeTest - one-stop quality open platform officially produced by Tencent. https://wetest.qq.com/.

[25] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation.. In *NDSS*.

[26] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[27] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[28] Kobra Khanmohammadi, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury. 2019. Empirical study of android repackaged applications. *Empirical Software Engineering* 24, 6 (2019), 3587–3629.

[29] Wenna Song, Jiang Ming, Lin Jiang, Han Yan, Yi Xiang, Yuan Chen, Jianming Fu, and Guojun Peng. 2021. App's Auto-Login Function Security Testing via Android OS-Level Virtualization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1683–1694.

[30] Yeali S Sun, Chien-Chun Chen, Shun-Wen Hsiao, and Meng Chang Chen. 2018. ANTSdroid: Automatic malware family behaviour generation and analysis for Android apps. In *Australasian Conference on Information Security and Privacy*. Springer, 796–804.

[31] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 1–41.

[32] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 165–176.

[33] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in android with {TIRO}. In *27th USENIX Security Symposium (USENIX Security 18)*. 1247–1262.

[34] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 358–369.

[35] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2018. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 814–828.

[36] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. Packergrind: An adaptive unpacking system for android apps. *IEEE Transactions on Software Engineering* (2020).

[37] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, 359–381.

[38] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*. Springer, 293–311.

[39] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.