# BPFDex: Enabling Robust Android Apps Unpacking via Android Kernel

Mingyang Li, Weina Niu, *Senior Member, IEEE*, Jiacheng Gong, *Graduate Student Member, IEEE*, Song Li, *Member, IEEE*, Mingxue Zhang, and Xiaosong Zhang, *Senior Member, IEEE*

***Abstract*—Malware developers exploit packing techniques to protect malicious apps from analysis. These evolving techniques, coupled with diverse anti-unpacker strategies, often render current studies ineffective in unpacking Android apps. In this study, we introduce BPFDex, a novel Android unpacking framework that leverages eBPF, a kernel component of the Android system. We successfully apply eBPF's excellent kernel observability and tracing capability to Android unpacking, both on real devices and emulators. Operating within the kernel space, BPFDex avoids drawbacks of common unpacking techniques. BPFDex monitors apps across both native and kernel layers, restores Dex data from memory, and adapts to different packing strategies according to observed packing behaviors. Furthermore, we summarize patterns in anti-unpacker behaviors among Android packers, establishing criteria to improve existing unpacking strategies. We conduct extensive experiments on BPFDex by leveraging more than 3k apps packed by over eight different packers. The results demonstrate that BPFDex successfully bypasses anti-unpacker strategies and unpacks apps packed by various packers, in contrast to other unpackers that can handle at most two packers.**

***Index Terms*—Android packer, android unpacking, dynamic analysis, android kernel.**

## I. Introduction

ANDROID's dominance in the dynamic landscape of mobile computing is evident, with over 70% market share [1]. And Android apps have long been a prime malware target for criminals aiming to gain unauthorized access to system resources, compromise data integrity, and violate user's privacy [2], [3], [5], [6], [7]. In recent years, attackers have abused packing techniques to bypass the vetting process and then distribute malicious code through app markets, targeting Android devices [4], [8]. Specifically, packing techniques introduce significant complexity when analyzing malicious code using static [48] and dynamic [51] analysis techniques. These techniques include detecting the running environment, hiding the original Dex data (i.e., the Dex file and its constituent items), and releasing them at runtime, etc [9], [10]. While packing services are initially intended to prevent legitimate apps from being reverse engineered and repackaged [11], the abuse of app packing has emerged as a new obstacle in safeguarding Android users.

Numerous unpacking tools (i.e., DexHunter) [12], have been developed for Android apps to address the aforementioned issue by revealing the concealed payload within packed apps. Nonetheless, the ongoing struggle between packers and unpackers persists, with packers continually adapting to undermine the effectiveness of unpackers. They employ a range of strategies to identify unpackers, hinder analysis, and alter the original Dex file loading process to prevent unpackers from gathering all concealed Dex data [10]. The current unpackers face several challenges that remain unaddressed:

- **C1-Effectiveness:** Unpackers are expected to retrieve all concealed Dex data and assemble it into a valid Dex file. However, the existing analysis and unpacking tools typically perform dynamic analysis based on Android emulators (e.g., Qemu) [13], [17], debugging techniques (e.g., ptrace) [14], [20], and DBI (dynamic binary instrumentation) hook frameworks (e.g., Frida and Valgrind) [15], [16], [18], [19]. These approaches may inadvertently leave unique traces in the system, such as special binary files and system properties, enabling sophisticated Android packers to identify them, thereby evading analysis (i.e., anti-unpacker). Furthermore, some packers [59], [60] may release the real code just before execution, subsequently erasing it. They may also hook system library methods (e.g., write) to hinder the dumping of dex code during the execution [10]. As a result, unpackers either fail to acquire all the concealed Dex data or end up with invalid Dex files.
- **C2-Adaptability:** Packers are constantly evolving, with each employing different packing strategies. Current unpackers often struggle to adapt due to rigid unpacking strategies and insufficient consideration of packers' evolution.

Received 2 February 2024; revised 12 April 2025 and 26 May 2025; accepted 25 July 2025. Date of publication 31 July 2025; date of current version 6 August 2025. This work was supported in part by the National Science Foundation of China under Grant 62372086, in part by the Natural Science Foundation of Sichuan under Grant 2025ZNSFSC0500, in part by Chongqing Natural Science Foundation Innovation and Development Joint Fund under Grant CSTB2023NSCQ-LZX0003, and in part by Sichuan Science and Technology Program under Grant 2024NSFTD0031. The associate editor coordinating the review of this article and approving it for publication was Dr. Qiben Yan. *(Corresponding author: Weina Niu.)*

Mingyang Li and Jiacheng Gong are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: wss-14@foxmail.com; gongjc.uestc@gmail.com).

Weina Niu and Xiaosong Zhang are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China, and also with the Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen 518110, China (e-mail: vinusniu@uestc.edu.cn; johnsonzxs@uestc.edu.cn).

Song Li and Mingxue Zhang are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China, and also with Hangzhou High-Tech Zone (Bin jiang) Institute of Blockchain and Data Security, Hangzhou 310052, China (e-mail: songl@zju.edu.cn; mxzhang97@zju.edu.cn).

Digital Object Identifier 10.1109/TIFS.2025.3594559

- **C3-Stability:** Unpackers should ensure minimal interference with the operation of the Android system and apps. Some existing unpackers depend on altering the Android system image or installing additional kernel modules [12], [19], [21], [22], [23] to achieve their objectives, leading to complex operations and an unstable system environment.

In this study, we aim to address the aforementioned challenges by developing BPFDex, a novel unpacking tool for Android devices. This tool leverages the Android kernel, built upon the Linux kernel. We utilize a recent Linux technology, eBPF (Extended Berkeley Packet Filter) [25], which is inherently available in the latest Android devices. The eBPF technology allows for the writing and compiling code in userspace, and subsequently injects it into an active kernel as a response to chosen triggers. To the best of our knowledge, BPFDex is the pioneering Android unpacker that employs eBPF, which fulfills the three challenges, to bypass anti-unpacker techniques and unpack more efficiently.

Specifically, unlike debuggers, which alter the status of the process being debugged, DBI that loads additional libraries and code into the application's memory, or emulators that possess distinct fingerprints, BPFDex leverages a Linux kernel tracing mechanism [26]. This mechanism operates within a virtual machine in the kernel, making it less susceptible to detection or interference by packers. Furthermore, BPFDex can effectively bypass the aforementioned anti-unpacker techniques and restore the original file from memory data released by packers during runtime, thereby addressing **C1**. To tackle **C2**, BPFDex doesn't rely on a static strategy for unpacking apps. Instead, it chooses suitable data collection points based on observed behaviors. Regarding **C3**, eBPF technology has been incorporated into the Android system kernel since Android 9.0 [43], which implies that, in contrast to other solutions [12], [18], [21], [22], [23], [75], BPFDex can monitor packers' behavior without tampering with the original running environment such as the system image, the application package, or the kernel. Moreover, the principle of safety and the verification process associated with eBPF ensure the correct function of BPFDex and the safety of Android kernel. Consequently, our method is robust and can operate safely on real devices, which effectively addresses **C3**.

More specifically, to enhance data collection and behavior monitoring, BPFDex performs monitoring in multiple layers including native libraries function calls and direct system calls in ART (Android runtime). For this purpose, we build a mapping from these functions to their exact addresses in the memory layout by resolving the loaded system libraries. Then, we utilize Kprobes (kernel space probes) and uprobes (user space probes) to hook these addresses for obtaining the concrete parameter values of respective functions at runtime. After analyzing the tracked information, we determine the packing behaviors and choose suitable Dex data collection points to gather the dynamically released Dex data. Lastly, we compile the retrieved Dex data into valid Dex files that can be analyzed using standard static analysis tools. Additionally, we formulate packers' features by integrating information from native libraries, classes and methods to improve efficiency of unpacking. If packers exhibit similar features, BPFDex can accelerate unpacking process by using the identical hooking strategy.

In summary, this paper makes the following key contributions:

- We propose a novel kernel-based approach to unpack the Android apps through packing behaviors tracking by using eBPF. This approach can effectively elude detection by packers, enhancing the efficiency of unpacking. Moreover, after inspecting more than eight packers using this approach, we have summarized patterns in anti-unpacker behaviors among Android packers, establishing criteria to help enhance the unpacking strategies of existing tools.
- We design and implement a new unpacking framework named BPFDex on both real devices and emulators. This framework dynamically tracks packing behaviors across multiple Android layers based on eBPF. BPFDex is released at https://github.com/BPFDex/BPFDex.
- We have conducted comprehensive experiments on BPFDex by leveraging more than 3k apps packed by over eight different packers. The evaluation results demonstrate that BPFDex successfully bypasses anti-unpacker strategies and unpacks apps packed by various packers, in contrast to other unpackers that can handle at most two packers.

The remainder of this paper is structured as follows: Section II provides essential background information. The design and implementation of BPFDex are detailed in Section III, while Section IV presents the experimental results. Section V discusses the limitations of BPFDex and future work. Finally, the related work is introduced in Section VI, and the paper concludes in Section VII.

## II. Background

### A. Extended Berkeley Packet Filter

The Extended Berkeley Packet Filter (eBPF) [25] is a lightweight virtual machine within the Linux kernel. It enables privileged users to safely load and execute bytecode in the kernel in response to selected events in both kernel and user space. An eBPF program can hook the kernel instruction using the kernel probe (kprobe) event or the kernel tracepoint event [28]. When the target instruction is executed, the corresponding eBPF program performs specified operations immediately, such as retrieving a register value or memory data. In addition to supporting kernel instrumentation, an eBPF program can hook user space programs through the user space probe (uprobe) event [28] or the userland statically defined tracing (USDT) event. Both kprobe and uprobe have been supported by the Android system since Android 9.0 [24]. The eBPF programs can be developed using the BPF Compiler Collection (bcc) [30], which provides high-level programming interfaces, to develop eBPF programs.

During eBPF's development, safety was paramount when considering its inclusion in the Linux kernel. All processes intending to load eBPF programs into the Linux kernel must operate in privileged mode (root), preventing untrusted programs from loading eBPF programs. Even when a process is permitted to load an eBPF program, all programs must still pass through the eBPF verifier [31], which ensures the

safety of the program itself. Furthermore, eBPF programs cannot directly access arbitrary kernel memory. Data and data structures outside the program's context must be accessed via eBPF helpers, ensuring consistent data access.

### B. Android App Packing

Typically, Android packers safeguard applications in three major ways: concealing Dex files, inhibiting the dumping of Dex files from memory, and obstructing static analysis.

*1) Concealing Dex Files:* The original Dex files in packed apps are typically concealed through runtime releasing, dynamic modification, and reimplementation using native code. Android packers primarily use native code to protect these Dex files, employing three main strategies.

Firstly, from version 5.0 onwards, Android's default runtime is ART. It performs on-device Ahead-Of-Time (AOT) compilation, compiling Dalvik bytecode into platform-specific native code during installation or initial run. This conversion uses the dex2oat function. Unpackers can directly access the hidden Dex files by decompiling the OAT files if dex2oat has processed the Dex files. To safeguard Dex data, packers often encrypt it and store it in special files. They then dynamically release the bytecode into memory for execution during the app's operation. Secondly, packers also protect Dex data by dynamically decrypting or modifying it before use and encrypting it after use. This prevents the Dex data from being dumped from memory. Consequently, specific Dex data might be missing in the dumped Dex data if it isn't collected at the correct points (i.e., functions) or at the right time. Thirdly, certain advanced packers (e.g., Qihoo [61]) reimplement the functionalities of specific methods using native code. As a result, there is no bytecode released for these protected methods during the execution of the apps.

*2) Inhibiting The Dumping Of Dex Files:* To prevent Dex files from being dumped form memory, packers typically employ four primary strategies: environment checking, anti-debugging, anti-DBI frameworks, and system library function hooking.

Firstly, packers scrutinize their operational environments to inhibit the execution of packed apps on emulators or customized AOSP devices. This is because many Android analysis and unpacking tools utilize the Android emulator or modify AOSP to facilitate unpacking. Secondly, to obstruct the debugging or instrumentation by unpacking tools, packers often attach threads to the packed app for anti-debugging and monitor process ports for anti-DBI frameworks. Thirdly, unpackers typically dump the Dex data from memory during runtime using Android's system library functions (e.g., *read* and *write*). Consequently, packers hook these functions to hinder the invocation of these methods.

However, BPFDex operates on actual devices and utilizes eBPF technology to hook the packed apps, subsequently extracting the Dex data via eBPF helpers. As a result, we successfully circumvented the aforementioned hurdles.

*3) Obstructing Static Analysis:* Packers a variety of techniques, including obfuscation and integrity checking, to safeguard Dex files from static tampering. Typically, Dex files are obfuscated prior to packing, implying that unpackers can only retrieve the obfuscated Dex data from memory during runtime. Analysts are then required to further deobfuscate the bytecode in the recovered Dex file to facilitate a better understanding of the app.

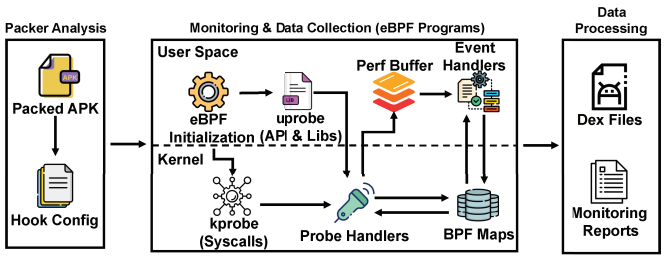


Fig. 1. The overview and workflow of BPFDex.

In addition, some packers verify the integrity of apps during installation or initial run to prevent tampering of apks by checking the MD5, SHA-1, and other signatures. If any discrepancies are detected between the original apk and the current one, the packers will terminate the process. Consequently, these technologies significantly escalate the cost of understanding and testing packed apps.

Nonetheless, the focus of this paper is on unpacking apps, thus deobfuscation and integrity checks bypass are beyond its scope.

## III. BPFDex

In this section, we introduce the overview and the workflow (III-A) of BPFDex. Then, we describe the components of BPFDex, including packer analysis module (III-B), monitoring and data collection module (III-C) and data processing module (III-D).

### A. Overview

Fig. 1 illustrates that BPFDex is comprised of three primary components: the packer analysis module, the monitoring and data collection module, and the data processing module. Utilizing Linux kernel features, BPFDex tracks packing behaviors and unpacks the app. It's worth noting that the kernel configurations necessary for enabling eBPF (e.g., CONFIG_KPROBES, CONFIG_UPROBES) have been activated since Android 9.0. Therefore, BPFDex doesn't necessitate any modifications to the Android system or the app under scrutiny. This architecture ensures minimal additional kernel overhead. In essence, the eBPF programs are solely tasked with monitoring app's behaviors and dumping data from memory, while packer recognition and data resolution are delegated to other modules operating outside the kernel.

Specifically, BPFDex first analyzes the packed apk to generate a hook configuration, which includes behavior monitoring points and data collection points (i.e., events) based on the apk's methods, classes, and native libraries. Subsequently, BPFDex launches the packed app and initializes eBPF programs with the hook configuration to execute tracing. The eBPF programs comprise a series of probe handlers (e.g., kprobs and uprobes) that are triggered by events specified in the hook configuration. While tracking the packer's behaviors, the eBPF programs extract data from the app's virtual memory and store it in a memory region within the kernel space.

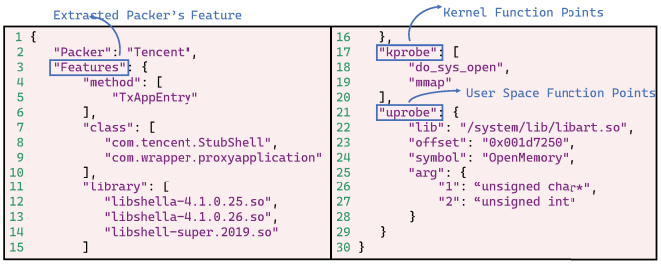


Fig. 2. A simplified example of the hook configuration.

Finally, the data processing module receives and processes the data from the eBPF programs to produce valid reassembled Dex files and monitoring reports, which include the packer's packing behaviors.

If malware gains root access, it can easily bypass BPFDex by exploiting kernel calls or loading malicious modules. However, in scenarios where an app lacks root privileges, it becomes significantly more difficult for malware to disable eBPF or detect BPFDex. Gaining root access is generally beyond the capabilities of typical apps, a presumption supported by Google's implementation of various strategies to secure the Android system's kernel [38], along with smartphone manufacturers' adoption of multiple techniques to reinforce the kernels of their customized Android systems [39], [40].

Theoretically, uprobes can be detected by identifying breakpoints within the `.text` segment of a library. However, this method is impractical for several reasons. Scanning for breakpoints requires a comprehensive examination of the `.text` memory section of the library file, necessitating continuous scanning of each instruction to identify breakpoints. This approach introduces substantial overhead, especially when an application must examine numerous library files. Moreover, if the scan is limited to only the start and end points of a function to minimize overhead, strategically placing breakpoints within the body of the function or inserting dummy instructions at either end can effectively conceal the breakpoints. An alternative detection method involves inspecting the `mmap`, but this can be bypassed by hiding the `mmap` file. Therefore, uprobes remain a robust method for unpacking, even in the face of these detection techniques.

### B. Packer Analysis

To unpack an app efficiently, BPFDex first analyzes the apk to identify the packer. For each specific packer, BPFDex determines monitoring and data collection points (i.e., functions) to track, generating a hook configuration with specific event (e.g., Dex parsing) information for eBPF programs. This means that instead of using static unpacking approaches, BPFDex can adapt to the evolution of packers by adopting different strategies for unpacking apps. Furthermore, for packers with identical features, the program uses the same hook configuration to expedite the unpacking process.

*1) Packer Recognition:* Despite the various methods packers use to pack apps, a packer typically employs a similar strategy to protect different apps, such as dynamically releasing bytecode and detecting emulators. This suggests that different apps have similar packing features in their methods, classes, and native libraries when packed by the same packer. Therefore, we generate features (i.e., $F$) for known packers by integrating packing features extracted from the packed apps, including methods ($f_m$), classes ($f_c$), and native libraries ($f_l$) (i.e., eq. 1). When analyzing a packed app, we identify the packers of the app by calculating the similarity (i.e., $S(F_a, F_p)$) between the packing features ($F_a$) and the features ($F_p$) of each known packer. In eq. 2, $F^*_m$, $F^*_c$, $F^*_l$ denote the features of a packer or app associated with methods, classes, and libraries respectively, while $w_*$ ($w_* > 0$ and $w_1+w_2+w_3$ equals 1.) represents the weight of an item, which can be tuned based on the significance of features. Note that these weights prioritize features for better packer characterization, not for differentiation among packers. Our experiments (See IV-B1.) demonstrate that features of native libraries are more important in packer recognition as packers iterate. If the similarity $S(F_a, F_p) > 0$, we consider the app as packed by packer $p$. If the app is not similar to any known packers, we classify it as an unknown packer. To ensure that the similarity between different packers is zero, we extract features with strong discriminative power. For example, Figure 2 (Lines 2-16) illustrates the features of the Tencent packer. These features consist of six items: one method, two classes, and three native libraries. For the method, "TxAppEntry" serves as the entry method for the packer, and the prefix "Tx" is derived from the Pinyin abbreviation of Tencent's Chinese name. This uniquely identifies the packer as belonging to Tencent. Similarly, the classes also reflect the packer's identity. The library names are unique to Tencent's packer, so we include these as part of the library features.

$$F = f_m \cup f_c \cup f_l \tag{1}$$

$$S(F^a, F^p) = w_1 \frac{|F^a_m \cap F^p_m|}{|F^p_m|} + w_2 \frac{|F^a_c \cap F^p_c|}{|F^p_c|} + w_3 \frac{|F^a_l \cap F^p_l|}{|F^p_l|} \tag{2}$$

To enhance the accuracy of packer recognition, we eliminate overlapping feature items to ensure feature uniqueness if they exist in multiple packers' features. Additionally, we dynamically update packers' features during unpacking to improve compatibility. Specifically, packers may have multiple versions with slight differences in implementation. When encountering an unknown packer, we manually identify it. If the features of the unknown packer belong to a known packer, we add the new features to the known packer. Otherwise, we consider the unknown packer as a new type and add its features to our database.

*2) Hook Configuration Generation:* We use hook configuration to leverage the programmability of eBPF. The hook configuration comprises a sequence of monitoring and data collection points (i.e., events) that allow eBPF programs to track the packed app. These events, which include two types - kprobe and uprobe, are defined according to the packing strategies. The kprobe is used to monitor system calls made by the packed app, while the uprobe detects the invocation of the Android framework API and the Android system's native libraries. Correct hook points enhance the efficiency and

effectiveness of unpacking, leading us to generate hook configurations that target packers with varying features. For instance, Fig. 2 shows a simplified hook configuration aimed at the Tencent packer. In this scenario, to monitor the packer's actions of opening files and mapping memory files, we employ the kprobe to hook the kernel functions, specifically *do_sys_open* and *mmap*. Additionally, we use the uprobe to hook the native function *OpenMemory* in `/system/lib/libart.so` at offset $0\times00d7250$ of the lib file. The *OpenMemory* function controls the parsing and loading of Dex files in the native layer, and it includes the memory address and size of the target Dex file as its first and second arguments, respectively. Therefore, we can hook the *OpenMemory* and extract these two arguments to dump the dex data from the memory.

To generate an effective hook configuration, we manually scrutinize the monitoring reports to filter the correct monitoring and data collection points. We eliminate points that yield little or no valid data to decrease the kernel payload produced by BPFDex. Moreover, when encountering an unknown packer, BPFDex will implement an initial hook configuration, which involves adding all possible events to the hook configuration.

### C. Monitoring and Data Collection

This module has three parts: initialization, monitoring and data collection. To execute monitoring and data collection, BPFDex first initializes the eBPF programs, including obtaining the addresses of functions and attaching the kprobe and uprobe to specified events involved in the hook configuration. Once specified events are observed, BPFDex proceeds to record packer's behaviors and collect data from the memory by using eBPF helpers.

*1) Initialization:* In the initialization phase, BPFDex parses the events from the hook configuration to enable the uprobe track the system native library functions. BPFDex achieves this by constructing a mapping between these functions and their address in the memory. To elaborate, BPFDex obtains the memory map information about the loaded system native libraries from the memory map (i.e., `/proc/pid/maps`) of the Zygote process. Subsequently, BPFDex disassembles the library files by using objdump [42] and extracts the symbols and offsets of the functions. In particular, BPFDex gets the file offset ($F_o$) of each function, the base virtual memory address ($V_b$) of Zygote process, and the virtual memory offset ($F_f$) of the native library files. Once all data is established, BPFDex calculates the virtual memory address ($V_e$) of each function using eq. 3. In the final step, BPFDex loads the eBPF programs to the kernel and attaches the kprobe to the raw syscalls and the uprobe to the events as per the established map. The eBPF programs can directly monitor raw syscalls by specifying kprobes' names, and track system native functions by attaching uprobes to the function addresses with their symbols.

$$V_e = V_b + F_f + F_o \tag{3}$$

Given the fact that eBPF programs operate within the kernel, it's important to ensure they don't cause any harm to it. This is where the eBPF verifier comes into play, meticulously examining eBPF programs before they're loaded into the kernel. The safety of these programs is ascertained through a two-step process. The first step involves a Directed Acyclic Graph (DAG) check, which disallows loops and conducts other Control Flow Graph (CFG) validations [44]. This step is particularly adept at detecting programs with unreachable instructions. Following this, the second step commences from the first instruction and explores all possible paths. It simulates the execution of each instruction, observing the changes in the state of registers and stack. If the eBPF programs fail to meet the safety requirements, the eBPF verifier steps in, halting the loading process to safeguard the kernel. This ensures that BPFDex poses no potential harm to the kernel.

*2) Monitoring:* Each eBPF program includes probe handlers that monitor kernel-level events with low overhead. Specific events can trigger these handlers, which run in the kernel. When particular events are observed, the probe handlers filter the event information to determine its relevance. When the event information relates to the app under analysis, the probe handlers process it according to predefined requirements, which include recording event details and collecting memory data. This ensures that no resources are wasted on irrelevant events. After processing all events, eBPF programs in the kernel send encoded data and a message through the perf buffer to notify event handlers to perform the necessary follow-up actions. The event handlers in user space receive the filtered event data from the kernel, parse and store it for subsequent analysis in the data processing module, and provide feedback to the probe handlers via BPF maps.

Since different apps share system native library functions, they may also trigger the observed events. This is irrelevant for unpacking the app and will make BPFDex wrongly treat behaviors of other apps as behaviors of the app under analysis. To address this, BPFDex gets the source (i.e., UID) of the calling process observed by the eBPF programs. Therefore, BPFDex can concentrate on monitoring behaviors and collecting data originating from the app under analysis, significantly reducing kernel overhead. In detail, the Android system assigns a unique UID to each user app during installation, which remains unchanged until the app is uninstalled. So BPFDex lets the eBPF programs record the UID value when specific events are observed by using *bpf_get_current_uid_gid*, an eBPF helper provided by Linux kernel, and stores the UID to the kernel memory.

In addition, since the packer calls a sequence of functions to release valid Dex bytecode, this sequence provides useful information for the data processing module to identify packing behaviors. Thus, BPFDex adds timestamps to the observed events information and sorts the events information in the data processing module according to their timestamps. In specific, the eBPF programs call the eBPF helper *bpf_ktime_get_ns* to get the running time when specific events are triggered and store it along with events information in the kernel memory.

*3) Data Collection:* Monitoring packers' behaviors is beneficial for understanding their mechanisms and identifying correct tracking points, but it's insufficient for unpacking apps. The data collection process retrieves Dex data from memory, which the data processing module later reassembles into Dex files. Packers utilize a series of system methods and functions, which contain significant Dex information as parameters, to parse and load the released Dex data. Therefore,

our focus is on capturing these parameter values of methods and functions invoked by the packed app. BPFDex directs the eBPF programs to retrieve memory data into kernel memory at runtime, primarily through two methods.

First, BPFDex employs *PT_REGS_PARM** [45], a series of macros provided by Linux kernel, to capture parameter values. For example, the second parameter of system native function *OpenMemory* is the size of a Dex file in the `unsigned int` type. Consequently, we use *PT_REGS_PARM2* to get the size information of Dex files when the function is invoked by the packed app. Second, if the parameter value we get is a base address of memory data, BPFDex fetches the data from this base address. Specifically, BPFDex uses *bpf_probe_read_user*, an eBPF helper provided by the kernel, to read memory data in user space starting from the based address. For instance, the first parameter of *OpenMemory* is the base address of a Dex file in the `unsigned char*` type. BPFDex first uses *PT_REGS_PARM1* to capture the parameter value, then relying on the base address and the file size, BPFDex uses *bpf_probe_read_user* to retrieve the in-memory Dex file when the packer invokes *OpenMemory*. Additionally, for Android framework methods, when methods at specific addresses are called, BPFDex captures the parameter values through the memory address according to the pre-built map.

For the kernel's safety and efficiency, the access amount of kernel memory for eBPF programs is limited to `MAX_BPF_STACK`, which is 512 bytes by default in the current Linux kernel version [25]. Unpacking apps requires substantial data transfer between kernel and user space, which exceeds the kernel memory limitation for eBPF programs. Therefore, BPFDex utilizes BPF maps, an interface provided by the Linux kernel, to store and read data between kernel and user space efficiently. The BPF map [46] stores data in a key-value format in the kernel without amount limitation, which can be shared by different eBPF programs in both kernel and user space.

### D. Data Processing

The data processing module of BPFDex accepts runtime data retrieved by eBPF programs for analyzing packing behaviors. This analysis is then clearly outlined in the subsequent monitoring report. Upon identifying specific packing behaviors, BPFDex modifies the monitoring and data collection points within the hook configuration of certain packers. This strategic adjustment allows BPFDex to adeptly adapt to the evolving nature of packers and enhances the overall efficiency and effectiveness of unpacking. Furthermore, BPFDex can reassemble Dex data collected from various points into valid Dex files, which are then verified through apk static analysis tools [20], [27], [47], [52].

*1) Anti-Unpacker Behaviors:* One of the main challenges faced by unpackers is the counteractions of packers. Packers prevent the dumping of Dex data and, upon detecting the presence of unpacking tools, shut down the apps to hinder unpacking. Therefore, BPFDex focuses on monitoring the anti-unpacker behaviors of packers to better understand their mechanisms and improve other unpacking tools. We conducted an analysis of more than 3k apps, which are provided by a prominent security company with a similar dataset size to other studies [18], [24], including custom packers and eight different commercial packers. We identified six types of anti-unpacker behaviors: emulator detection, anti-debugging, anti-DBI frameworks, system library hooking, time delay checking, and root detection. As summarized in Table I, we list the monitored methods and functions used to identify anti-unpacker behaviors. The notation **PARM*** indicates that BPFDex records the *th parameter value of the called function, while methods and functions without **PARM*** denote that BPFDex only records their invocation. BPFDex determines if a packer exhibits specific behavior based on whether the parameter value satisfies certain conditions or if certain functions are called. It's important to acknowledge the continuous evolution of packers, which complicates the comprehensive collection of all anti-unpacker behaviors and identification criteria. But we have endeavored to gather as many as possible.

①**Emulator Detection (EUD):** Packers often prevent themselves from running in emulators, which are commonly used for dynamic analysis of Android apps (e.g., QEMU). To achieve this, packers compare system information between real devices and emulators. Emulators often have unique device fingerprints—like fixed phone numbers, X86 CPU architecture, or unknown device vendors—which packers identify by calling Android framework APIs (e.g., *TelephonyManager*) to fetch device hardware information. Packers also examine the existence of emulator-specific system files (e.g., `/proc/tty/drivers` and `/system/bin/qemu-props`).

**Identification Criteria:** BPFDex tracks the app's calls to system information functions and evaluates library function (e.g., *open*, *strstr*) parameters to discern if the packer inspects system properties' values to detect the emulator.

②**Anti-Debugging (ADB):** Packers usually call *isDebuggerConnected* to detect debuggers. And since a process can be attached by only one another process at a time, packers invoke *ptrace* to determine whether the app is attached by other process. Moreover, packers detect the presence of debugger via examining process status of the app and default debugger TCP port by checking system files (e.g., `/proc/self/status` and `/proc/net/tcp`).

**Identification Criteria:** BPFDex checks if the app has called *isDebuggerConnected* (*gDebuggerConnected* in native) or *ptrace*. It also checks if the app has read process information to search for a debugger by examining library function parameters (e.g., *open* and *strstr*).

③**Anti-DBI Frameworks (ADF):** DBI frameworks [15], [16], [53] are commonly used to instrument Android apps. Packers inspect suspicious installed packages by invoking *PackageManager* APIs and comparing installed packages with common DBI frameworks. Given that DBI frameworks introduce artifacts (e.g., jar files) to memory, packers examine the memory map through system files (e.g., `/proc/pid/maps`) to locate files associated with DBI frameworks. Additionally, as some DBI frameworks open specific TCP ports to connect to DBI servers, packers check the TCP port state through system files (e.g., `/proc/net/tcp`) to match default DBI framework ports.

**Identification Criteria:** BPFDex identifies suspicious values (e.g., frida) in system function parameters (e.g., *open* and *strstr*) that may indicate DBI framework fingerprints.

TABLE I
THE RULES OF IDENTIFYING ANTI-UNPACKER BEHAVIORS

| Anti-unpacker Type | Monitored Methods and Functions | Behaviors |
|---|---|---|
| Emulator Detection | *libc.__system_property_get* (or *libc.__system_property_read*) | Obtaining device's information and comparing it with special values. |
| | *libc.open* (**PARM1**) | |
| | *libc.strstr* (**PARM2**) | |
| Anti-Debugging | *libart.gDebuggerConnected* | Determining whether the app is being debugged. |
| | *libc.ptrace* | Determining whether debuggers attach to ptrace. |
| | *libc.open* (**PARM1**) | Checking debugger processes and status of apps via system files. |
| | *libc.strstr* (**PARM2**) | |
| Anti-DBI Frameworks | *libc.open* (**PARM1**) | Detecting DBI frameworks by inspecting installed packages and runtime context. |
| | *libc.strstr* (**PARM2**) | |
| System Library Hooking | *libc.dlsym* (**PARM2**) | Locating actual memory address of the target function. |
| | *libc.mprotect* (**PARM1** & **PARM2**) | Making code segments writable to enable hook. |
| Time Delay Checking | *libc.gettimeofday* (or *libc.time*) | Checking additional delay introduced by dynamic analysis. |
| Root Detection | *libc.execvp* (**PARM1** & **PARM2**) (or *libc.execvpe*) | Executing commands that require root privileges like *su*. Executing *mount* to check writable access of mounted system directories. |
| | *libc.__system_property_get* (or *libc.__system_property_read*) | Checking device properties like `ro.debuggable` `ro.secure` and `ro.build.tags`. |
| | *libc.access* (**PARM1**) (or *libc.open* (**PARM1**)) *libc.strstr* (**PARM2**) | Matching rooting apps, and root binaries. |

[1] The notation **PARM*** indicates that BPFDex records the *th parameter value of the called function, while methods and functions without **PARM*** denote that BPFDex only records their invocation.
[2] The functions *strcmp*, *strcasecmp*, *strncmp*, *strncasecmp*, and *strstr* share similar functionalities, enabling them to be interchangeable.
[3] The functions *open*, *openat*, *fopen*, *freopen*, and *fdopen* possess similar functionalities, allowing for interchangeability among them.

④**System Library Hooking (SLH):** Unpackers generally use system library functions (e.g., *open* and *write*) to analyze packed apps and extract Dex data. Packers can prevent this by hooking related functions in system libraries (e.g., `libart.so` and `libc.so`) using GOT/PLT hooking or inline hooking [10].

**Identification Criteria:** If packers attempt to hook system library functions, they typically locate the symbols of the target functions in memory and modify the access permissions of the memory region of system libraries. BPFDex tracks parameter values of *dlsym* to determine if the app is trying to locate sensitive functions (e.g., *write*). It also monitors memory region values of the *mprotect* function, controlling memory regions' access permissions, to check if the region involves system libraries.

⑤**Time Delay Checking (TDC):** Packers may infer they are being analyzed through dynamic analysis tools if they detect additional time delays. Dynamic analysis slows down app execution, and packers calculate the time spent executing a specific task to detect these delays [54].

**Identification Criteria:** Packers typically invoke system library functions (e.g., *gettimeofday*) at least twice to get the start and end times of a specific task. BPFDex monitors these function invocations to check if they are called consecutively. Additionally, we append additional seconds to the return value during the second invocation. Should the app crash, this accounts for any potential time delays within the app.

⑥**Root Detection (RDT):** Many unpackers necessitate root access to facilitate unpacking via hooking or other strategies. Consequently, packers frequently employ various techniques to avoid operating on rooted devices. A common method involves executing shell commands. For example, packers verify the execution capability of the *su* command, which typically resides in rooted systems. Packers also scrutinize the permissions of mounted system directories (e.g., `/system/bin`) that may be writable on rooted devices. This is done by invoking the *mount* command to inspect directory access. Moreover, packers examine distinct system properties (e.g., `ro.build.tags`) that signify whether the systems are rooted. They further detect a root environment by identifying rooting apps (e.g., magisk) using *PackageManager* APIs, and verifying the existence of root binaries (e.g., `/system/xbin/su`).

**Identification Criteria:** To identify shell commands, BPFDex monitors *exevcp* calls to discern whether the command includes sensitive commands and suspicious parameters. Given that the library function *__system_property_get* and *__system_property_read* can directly fetch rooted system properties, BPFDex watches for such invocations to detect root detection behaviors. Furthermore, BPFDex tracks *access*, and *strstr* to inspect the matching of rooting apps and root binaries.

*2) Unpacking Apps:* In order to retrieve Dex files from packed Android apps, BPFDex firstly gathers dynamically released Dex data from pre-set data collection points as specified in the hook configuration. Secondly, it reassembles Dex items to form Dex files and repairs the obtained Dex files. The validity of these Dex files is then confirmed by decompiling them using available static analysis tools.

①**Dex Collection Points:** Given that all Dex data must be parsed by ART before use, we designate key Dex data resolution points for BPFDex as our default Dex data collection points. These include Dex file parsing, class defining, method resolving, and method interpreting stages of the entire Dex data loading process. As outlined in Table II, BPFDex gathers

TABLE II
THE DEX DATA COLLECTION POINTS AND THEIR CORRESPONDING EVENTS

| Category | Methods and Functions | Events |
|---|---|---|
| Dex Metadata | *DexFile::DexFile* | Loading and parsing Dex files. |
| | *DexFile::Open* | |
| | *DexFile::OpenMemory* | |
| | *DexFile::OpenFile* | |
| Dex Items | *ClassLinker::DefineClass* | Defining classes. |
| | *ArtMethod::LoadMethod* | Resolving and linking methods. |
| | *ClassLinker::LinkCode* | |
| | *ExecuteSwitchImpl* | Interpreting methods. |
| | *ExecuteGotoImpl* | |

two types of Dex data from memory: the metadata (e.g., size and address) of in-memory Dex files and the Dex items (e.g., classes and methods) of Dex files. We group these Dex data collection points by their processed Dex data type and related events.

To load Dex files, ART first parses the target files, each referred to by a `DexFile` object containing the Dex file's metadata. We can therefore collect related information when ART parses Dex files. Specifically, ART employs *DexFile::OpenFile* to open file system Dex files and *DexFile::Open*, *DexFile::OpenMemory*, and *DexFile::DexFile* to parse Dex files mapped into memory. BPFDex, in turn, monitors these functions to gather Dex files' metadata and retrieves Dex files from memory based on this metadata.

Recognizing that many unpackers use a one-pass unpacking strategy to retrieve dynamically released Dex data, BPFDex collects scattered Dex items for further reassembling into Dex files. This is because packers may replace original Dex items (e.g., `class_def_item` and `code_item`) with blank data and dynamically recover them or modify ART runtime objects (e.g., `Class` and `ArtMethod`) to evade unpacking. According to ART runtime, ART uses *ClassLinker::DefineClass* to create `Class` objects that represent classes in Dex files, and for methods in each class, ART utilizes *ArtMethod::LoadMethod* and *ClassLinker::LinkCode* to resolve and link them. Post parsing, ART employs *ExecuteSwitchImpl* or *ExecuteGotoImpl* (i.e., the goto-based and switch-based interpreters) to interpret the bytecode of methods stored in `CodeItem` objects. We thus select these methods and functions as BPFDex's default Dex item collection points.

For a new packer, BPFDex defaults to collecting data from all Dex collection points. If no valid data are observed at specific collection points, BPFDex removes those points from the packer's hook configuration.

②**Dex Files Reassembling and Repairing:** To reassemble Dex items into valid Dex files, BPFDex first logs the memory address of the dumped Dex items and Dex files that are used by ART to generate runtime objects. Post data collection, BPFDex calculates each Dex item's file offsets based on the memory address relation between the Dex files and Dex items. BPFDex then fills the dumped Dex files with Dex items, locating them through file offsets.

Moreover, many packers alter the header of in-memory Dex files to hinder searching or decompiling. As such, BPFDex repairs the header of obtained Dex files (e.g., magic number, checksum, and signature) according to their content.

After reassembling and repairing the Dex files, BPFDex employs Apktool to validate the output Dex files. If the output Dex files can be successfully decompiled, we consider them valid.

## IV. EVALUATION

We implemented BPFDex utilizing approximately 7k lines of C/Python code and 2k lines of Java code, leveraging eadb [50], an eBPF Android debug bridge which grants interfaces for accessing eBPF functionalities on Android devices; bcc [30], a toolkit for creating efficient kernel tracing and manipulation eBPF programs. We deployed BPFDex on two platforms: a Pixel 6 device, running Android 13.0 with the 5.10.107 Android kernel, and an Android emulator, functioning on Android 7.1 with the 5.15.0 Linux kernel. We choose a recent version and an older version to demonstrate the versatility and compatibility of BPFDex across different generations of Android.

In the following section, we aim to evaluate BPFDex's proficiency and performance. We have formulated the subsequent research questions to guide our assessment:

- **RQ1:** Does BPFDex accurately identify popular commercial packers and generate hook configurations for a specific packer?
- **RQ2:** Can BPFDex effectively restore the original Dex files from packed apps and outperform other existing unpacking tools?
- **RQ3:** Can BPFDex enhance other unpacking tools by exposing the anti-unpacker strategies of packers?
- **RQ4:** Does BPFDex aid malware detection by facilitating static analysis tools?
- **RQ5:** Is the overhead introduced by BPFDex within acceptable limits?

### A. Data Set

We evaluate BPFDex using two distinct data sets. The first set (D-1) comprises 240 packed apps accompanied by the corresponding Dex files. Initially, we randomly downloaded 40 different open-source apps from F-Droid [56], none of which were packed. To verify BPFDex's effectiveness at different times, we divided the 40 apps into two equal parts. We uploaded the first 20 apps to VI popular online commercial packing service vendors (i.e., Ali [57], Baidu [58], Bangcle [59], Ijiami [60], Qihoo [61], and Tencent [62]), which are commonly used in research [8], [10], [18], [29], in Nov. 2022 (P-22) and uploaded the remaining 20 apps in Jul. 2023 (P-23). This yielded 240 packed apps, which is similar in size to other studies [10]. We only used these 6 packing services because several online commercial packing service providers (e.g., Kiwi [63], Naga [64], Testin [65]) discontinued their public packing services after September 2022.

The second set (D-2) comprises 1,123 malware samples, which were packed by eight known packers (i.e., Ali, APKProtect [66], Baidu, Bangcle, Ijiami, Kiwi, Qihoo, Tencent) and custom packers, supplied by a prominent security company. It's noteworthy that these samples, mostly gathered between 2018 and 2023, may be packed by multiple versions of the same packer.

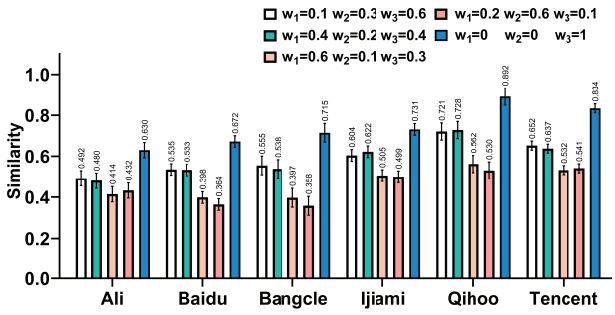


Fig. 3. The similarities between different versions of each packer at different weight settings.

## B. Packer Analysis

In this experiment, we apply apps that are packed by various commercial packers to evaluate whether BPFDex can effectively identify packers and then generate a corresponding hook configuration.

*1) Packer Recognition:* Initially, we select an app packed by six commercial packers from both P-22 and P-23 to generate features for each commercial packer. Subsequently, we cross-recognize apps in P-22 and P-23. In other words, we use features generated from P-22 to recognize packers in P-23, and vice versa. Furthermore, we manually label the packers of 200 apps in D-2 and employ BPFDex to identify these apps using features generated from P-22 and P-23. We conduct this experiment using the mechanism outlined in III-B with a series of distinct feature weight settings.

For clarity, we present the results of five distinct feature weight settings to illustrate the general trend in similarity between different version of a same packer. The experimental results depicted in Fig. 3 demonstrate that BPFDex can effectively identify these six commercial packers. Additionally, the similarities between different versions of each packer vary significantly at different feature weight settings. Generally, the similarities increase when the native library feature has a higher weight but decrease when the opposite is true. This suggests that the native libraries are instrumental in recognizing packers. Packers typically implement packing code by introducing additional native libraries to the target apps, and the entry classes and methods for invoking these libraries are often concealed or obscured. Moreover, packers tend to modify packing methods and classes but leave native library features intact across different versions. Importantly, the similarities between different packers are zero, indicating that we have extracted unique features for each packer.

*2) Hook Configuration Generation:* We evaluate BPFDex's ability to correctly generate the hook configurations using the latest packers in P-23. The primary task of generating hook configuration involves identifying data collection points for specific packers. According to our experiment, all six packers execute their code in the native layer and dynamically release the Dex data into memory at runtime. Ali, Baidu, and Qihoo dynamically release valid Dex files before invoking the function *DexFile::DexFile*, while Bangcle and Tencent do so prior to calling the function *DexFile::OpenMemory*. Therefore, BPFDex tracks these functions as data collection points for the aforementioned packers. Conversely, Ijiami releases Dex files with blank Dex items when creating the `DexFile` objects, subsequently modifying the `CodeItem` before the ART resolves methods. As such, BPFDex tracks *DexFile::DexFile* and *ArtMethod::LoadMethod* as data collection points for Ijiami. The experimental result shows that BPFDex can accurately generate hook configurations based on the packer recognition result.

**Answer to RQ1:** BPFDex can effectively identify popular commercial packers in packed apps and generate a corresponding hook configuration for the app under analysis.

## C. Recovering Dex Files

To assess the effectiveness of BPFDex in retrieving Dex files, we examined its performance on apps from both D-1 and D-2 datasets. Our findings are then compared with results from other open-source unpackers.

*1) Unpacking Outcomes:* With the source apps of six commercial packers in D-1 at our disposal, our first step was to deploy BPFDex on the D-1 apps. This allowed us to compare the recovered Dex files with the original ones, gauging BPFDex's efficiency. Post-unpacking, a suite of static analysis tools, namely Baksmali [52], IDA Pro [20], Jadx [27], and JEB [47]—each using distinct Dex file verification methods—was employed. The goal was to authenticate the recovered Dex files and to discern the disparities between the decompilation outcomes of the original and recovered Dex files.

Our trials revealed that BPFDex adeptly retrieves valid Dex files from apps safeguarded by notable commercial packers, including the likes of Ali, Baidu, Bangcle, and Tencent. For apps fortified by Ijiami, BPFDex's capability was restricted to retrieving Dex data from executed methods. This limitation stems from the packer's dynamic release of Dex data in method granularity, altering `code_item` during method execution. Furthermore, the Qihoo packer's method of reconstituting app functions using native code means that BPFDex can only retrieve the Dex files that were not converted to native code. It's pivotal to highlight that this issue remains unsolved by any existing unpacker.

Expanding our evaluation, BPFDex was applied to the 1,123 apps in D-2. With the assistance of various static analysis tools, the integrity of the recovered Dex files was confirmed. BPFDex displayed competence in effectively unpacking all apps, with every Dex file being seamlessly disassembled.

*2) Comparison:* A parallel investigation was executed using three renowned open-source unpackers from both the research and industrial sectors: BlackDex [33], DexHunter [12], and FRIDA-DEXDump [35]. Table III captures the comparative results, revealing that these unpackers faced myriad challenges. Specifically, BlackDex struggles to unpack apps protected by Bangcle, Ijiami, and Qihoo, due to targeted process checks, and also falters with apps from Tencent because of inappropriate Dex data collection timing. FRIDA-DEXDump is hindered by DBI framework checks when unpacking apps from Bangcle and Ijiami. It also faces issues with apps from Qihoo and Tencent due to ill-timed Dex data collection. DexHunter's methodology, which involves embedding

TABLE III

UNPACKING RESULTS COMPARISON OF BPFDEX, BLACKDEX, DEXHUNTER AND FRIDA-DEXDUMP

| Unpacker \ Packer | Ali | Baidu | Bangcle | Ijiami | Qihoo | Tencent |
|---|---|---|---|---|---|---|
| **BPFDex** | ✔ | ✔ | ✔ | ✔[1] | ✔[2] | ✔ |
| BlackDex[33] | ✔ | ✔ | ✖ | ✖ | ✖ | ✖ |
| DexHunter[12] | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ |
| FRIDA-DEXDump[35] | ✔ | ✔ | ✖ | ✖ | ✖ | ✖ |

[1] The packer Ijiami only releases the Dex data of methods that need to be executed, thus BPFDex can only recover these Dex data.

[2] The packer Qihoo reimplements methods of packed apps through native code, thus these Dex data can't be recovered because they will never be released into the memory.

TABLE IV

DETECTION OF ANTI-UNPACKER BEHAVIORS IN COMMERCIAL PACKERS AND THEIR PROPORTION IN D-2

| | Ali | Baidu | Bangcle | Ijiami | Qihoo | Tencent | Ratio in D-2 |
|---|---|---|---|---|---|---|---|
| EUD | ✖ | ✔ | ✖ | ✔ | ✖ | ✖ | 248/1123 (22.1%) |
| ADG | ✖ | ✔ | ✔ | ✔ | ✔ | ✖ | 475/1123 (42.3%) |
| ADF | ✖ | ✔ | ✔ | ✔ | ✔ | ✔ | 217/1123 (19.3%) |
| SLH | ✖ | ✖ | ✔ | ✔ | ✖ | ✖ | 387/1123 (34.5%) |
| TDC | ✖ | ✖ | ✖ | ✔ | ✔ | ✖ | 269/1123 (24.0%) |
| RDT | ✖ | ✖ | ✔ | ✔ | ✖ | ✖ | 292/1123 (26.0%) |

unpacking code into runtime functions by directly amending AOSP, could only penetrate apps shielded by Ali.

**Answer to RQ2:** BPFDex exhibits proficiency in recovering Dex files from packed apps, outshining other unpackers such as BlackDex, DexHunter, and FRIDA-DEXDump in performance.

### D. Discovering Anti-Unpacker Behaviors of Packers

Using BPFDex, we identify the anti-unpacker behaviors of the apps in D-1, as summarized in the left section of Table IV. The "✗" symbol indicates that the behavior was not detected in the packer, while the "✓" symbol signifies that the behavior was detected. Additionally, the right section presents the ratio of anti-unpacker behaviors detected by BPFDex for the malicious apps in D-2.

*1) Commercial Packers:* Both Baidu and Ijiami packers employ emulator detection, thwarting analysis in emulators. Their detection relies on inspecting system property values such as `ro.product.model`. Ijiami further examines specific files, such as `/proc/tty/drivers`, to detect Qemu.

Anti-debugging mechanisms are prevalent in Baidu, Bangcle, Ijiami, and Qihoo packers. Baidu and Ijiami, for instance, utilize *Debug.isDebuggerConnected* to identify JDWP-based debuggers [67]. Bangcle and Qihoo, on the other hand, refer to the `/proc/self/status` system file, relying on the `TracerPid` value to determine debugging status.

Several packers, including Baidu, Bangcle, Ijiami, Qihoo, and Tencent, incorporate anti-DBI techniques. They probe the `/proc/self/maps` system file to identify files like `frida-agent-32.so`, `edxp.jar`, and `app_process32_xposed` mapped in memory. Additionally, Qihoo scans installed apps to pinpoint DBI frameworks and decompiles target apps' dex files in search of suspicious Java classes.

Packers Bangcle and Ijiami utilize system library hooking. Bangcle, for instance, intercepts system functions such as *read* and *write* from `libc.so` to secure Dex data. Concurrently, Ijiami hooks *__android_log_buf_write* from `liblog.so` to prevent unauthorized log access.

Ijiami and Qihoo packers implement time delay checks in their code to monitor the execution time of specific native tasks, capping it at one second.

Root detection mechanisms are evident in both Bangcle and Ijiami. They inspect system properties, like `ro.debuggable`, using *__system_property_get*. Notably, Ijiami examines paths related to rooting apps and checks for specific processes, including `su` and `su_daemon`.

*2) Anti-Unpacker Behaviors in Malicious Apps:* In our analysis of D-2's malicious apps, all six anti-unpacker behaviors were identified. Anti-debugging and system library hooking emerged as predominant. However, while many apps leverage root detection, time delay checks, and emulator detection, the use of anti-DBI frameworks is less frequent among them.

**Answer to RQ3:** BPFDex is proficient in uncovering the anti-unpacker tactics of packed apps, thereby aiding unpackers in adjusting to the evolving defenses of packers.

TABLE V

COMPARISON OF SENSITIVE API CALLS IN PACKED MALWARE VERSUS RECOVERED DEX FILES

| Packer | Ali | APKProtect | Baidu | Bangcle | Ijiami | Kiwi | Qihoo | Tencent | Custom |
|---|---|---|---|---|---|---|---|---|---|
| Numer of apps | 29 | 2 | 63 | 38 | 40 | 2 | 389 | 518 | 20 |
| *PA* in avrage | 0.00 | 5.50 | 0.87 | 1.82 | 0.25 | 9.00 | 0.58 | 1.22 | 2.75 |
| *RA* in avrage | 18.85 | 9.50 | 15.66 | 8.47 | 10.10 | 16.00 | 16.71 | 13.19 | 8.89 |
| Difference | +18.85 | +4.00 | +14.79 | +6.65 | +9.85 | +7.00 | +16.13 | +11.97 | +6.14 |

### E. Enhancing Static Analysis With BPFDex

To assess the effectiveness of BPFDex in enhancing static analysis, we started by analyzing the 1,123 malware in D-2, taking these results as our reference point. Subsequently, we utilized BPFDex to unpack the malware, performing the same static analysis on the extracted Dex files. It's worth noting that numerous static analysis tools [68], [69], [70], [71], [72] identify malicious apps by detecting sensitive API calls. As such, we employed Androguard, a static analysis tool proven effective in other studies [41], [49], to determine the number of sensitive API calls in both the packed malware and the recovered Dex files.

Table V displays the findings, with *PA* and *RA* denoting the count of sensitive API calls in packed apps and the related recovered Dex files, respectively. The data reveals that the recovered Dex files of the malicious apps, after unpacking with BPFDex, unveil a higher number of sensitive API calls. This has potential implications for improving the precision of Android malware detection. To illustrate, while malware packed by entities like Ali, Baidu, Ijiami, and Qihoo initially exhibit few to no sensitive APIs, a significant number emerge in the associated recovered Dex files.

**Answer to RQ4:** BPFDex enhances static analysis tools for malware detection by revealing concealed details in Dex files.

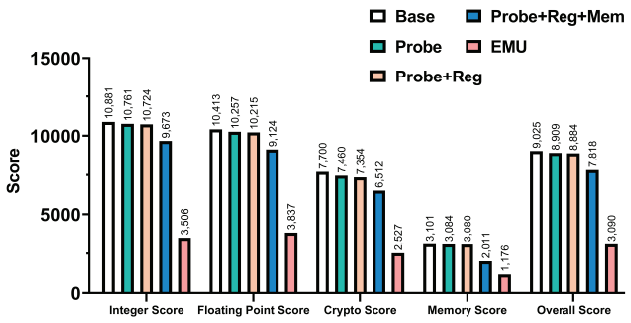


Fig. 4. Geekbench results (high score means high performance).

### F. Overhead

The overhead from BPFDex primarily stems from three operations: eBPF probe tracing, register access, and memory data reading. To measure the impact of these operations, we configure BPFDex to execute either a single operation or a combination of these operations within a single test. Note that register access and memory data reading cannot be performed in isolation. Since many dynamic analyses and unpacking procedures rely on emulators, we compare a QEMU-based Android emulator running Android 7.1 with BPFDex, both running on the same host machine. In our evaluation, we execute the benchmark tests provided by Geekbench 2 [73] ten times. Our focus is on the average scores for integer, floating point, crypto, and memory, as well as the overall scores determined by Geekbench. All tests are conducted on a Linux host with an Intel i5-10400 CPU and 16GB RAM.

The results from Geekbench are illustrated in Fig. 4. We use the scores from BPFDex with all functions disabled as our baseline (represented by the *Base* bar). Activating the probe tracing function shows minimal impact on performance, as represented by the *probe* bar. This trend is consistent when the register accessing function is added (i.e., the $Probe + Reg$ bar). However, introducing the memory data reading function (represented by the $Probe + Reg + Mem$ bar) results in a roughly 10% slowdown. In contrast, the emulator's performance, which reaches only about a third of the *Base* score, is represented by the *EMU* bar.

**Answer to RQ5:** Utilizing kernel characteristics, BPFDex introduces minimal overhead when tracking and monitoring apps.

## V. Discussion

The limitations of BPFDex are primarily fourfold.

The requirement for root privileges to load eBPF code into the Android kernel is a constraint. As a workaround, integrating BPFDex within the system image could eliminate the need for device rooting, since BPFDex operates autonomously without additional permissions. Moreover, BPFDex's kernel-space execution means it remains undetectable to apps lacking root access. It is important to note that our framework is intended for research purposes, particularly in controlled environments like malware detection and analysis. We do not recommend its deployment in production environments, such as Google's Android ecosystem, where security, stability, and performance considerations are paramount. Therefore, while our framework offers value in specific research contexts, we do not advocate for its widespread adoption in end-user devices.

BPFDex's reliance on dynamic analysis presents an inherent challenge: it may not execute all instructions of the target apps, potentially overlooking unexecuted Dex data. This issue, often inadequately addressed in other studies [36], [37], [75], typically involves alterations to the AOSP, leading to significant system overhead and increased crash probability. To address this, future iterations will incorporate automated app testing tools, like Monkey [74], to ensure more comprehensive app execution.

The evolving nature of packers and their corresponding anti-unpacker tactics also poses a risk of evasion, which is a challenge inherent to all packers. We plan to enhance BPFDex's detection capabilities by amassing a broader set of rules and employing machine learning algorithms to recognize emerging patterns.

BPFDex currently does not monitor Java methods or the parameters of the Android framework API, as this requires deconstructing Java class data in memory. This aspect is designated for future investigation.

## VI. Related Work

The dynamic nature of Android packers, designed to obfuscate malicious payloads, has prompted extensive research [12], [18], [21], [22], [23], [75] and industrial efforts [33], [34], [35], [36], [37]. Yet, the continual evolution of packers outpaces the static unpacking strategies employed by tools like DexHunter [12] and AppSpear [22], leading to their diminished effectiveness. Solutions such as ReDex [75] impose substantial overhead by executing each app method comprehensively. Others, including Packergrind [18], DexX [21], FRIDA-DEXDump [35], dumpDex [34], adapt to the changes in packers to some degree but depend on detectable techniques like Dynamic Binary Instrumentation (DBI), Virtual Machine Introspection (VMI), or system modifications, making them vulnerable to sophisticated anti-unpacker defenses. DeepAutoD [76] innovatively integrates a deep learning model into the malware detection process after unpacking. However, it still relies on system image modification to perform the unpacking of apps. In contrast, BPFDex can effectively bypass detection of packers and adjust unpacking strategies according to observed packing behaviors.

Parema [29] innovatively addresses the challenges posed by VM-based Android packers by introducing a learning-based unpacking framework capable of reconstructing original code semantics. Nevertheless, its effectiveness relies on the availability of representative training samples or access to source-level virtual machine semantics. Moreover, since it specifically targets VM-based packers, it is not applicable to conventional packing techniques.

Hardware-assisted methods have also been explored for enhanced program debugging and tracing on both x86 and ARM architectures. They utilize advanced features such as the Performance Monitoring Unit (PMU) and Embedded Trace Macrocell (ETM). For example, Ninja [55] employs TrustZone alongside PMU and ETM for secure amd transparent application tracing. Similarly, hardware-assisted tracing has been applied to Android unpacking, with Happer [10] utilizing ETM

TABLE VI
COMPARISON OF RELATED WORK

| Category | Tool/Method | Focus/Approach | Strengths | Weaknesses |
|---|---|---|---|---|
| Proposed Method | **BPFDex** | Dynamic unpacking (eBPF) | Effective unpacking and adaptation | Depends on eBPF for dynamic unpacking |
| Static Unpacking Strategy | DexHunter | Static unpacking | Focused on static analysis | Limited by static unpacking methods |
| | AppSpear | Static unpacking | Focused on static analysis | Limited by static unpacking methods |
| System Modification | DexX | Dynamic unpacking | Adapts to changes in packers | Depends on DBI, VMI, system modifications |
| | ReDex | Dynamic execution | Comprehensive app method execution | Imposes substantial overhead |
| | DeepAutoD | Dynamic unpacking | Integrates deep learning for malware detection | Depends on DBI, VMI, system modifications |
| DBI | FRIDA-DEXDump | Dynamic unpacking | Unpacking apps with dynamic instrumentation | Vulnerable to sophisticated anti-unpacker defenses |
| | Packergrind | Dynamic unpacking | Adapts to changes in packers | Depends on DBI, VMI, system modifications |
| | dumpDex | Dynamic unpacking | Unpacking apps with dynamic instrumentation | Vulnerable to sophisticated anti-unpacker defenses |
| VM-based Packer | Parema | Learning-based unpacking for VM-based Android packers | Effectively recovers code semantics from VM-protected apps | Not suitable for standard (non-VM) packers |
| Hardware-based Analysis | Ninja | Hardware-assisted tracing (TrustZone, PMU, ETM) | TrustZone for secure tracing | Efficiency bottlenecks due to ETM stream analysis |
| | Happer | Hardware-assisted tracing (ETM) | ETM for tracing packing activity | Network security risks due to ETM stream analysis |
| eBPF-based Analysis | NCScope | Real-time detection with eBPF | Real-time malware detection with eBPF | Limited to real-time detection |
| | BPFroid | Real-time detection with eBPF | Real-time malware detection with eBPF | Limited to real-time detection |

to observe packing activity and adaptively select unpacking strategies. However, Happer's dependence on ETM stream analysis via an online host computer introduces efficiency bottlenecks and network security risks. BPFDex can run and process data on real devices, thus avoiding complex data transfers and improving the efficiency of unpacking.

The eBPF technique has recently been integrated into Android malware analysis. Tools like NCScope [24] and BPFroid [32] leverage eBPF for real-time detection and analysis of malicious native code. Contrasting these applications, BPFDex innovatively applies eBPF to the domain of Android unpacking, analyzing packers and collecting runtime data to extract hidden Dex files and identify anti-unpacker tactics. Notably, BPFDex pioneers the use of eBPF in the field of Android unpacking.

Comparison of related work is shown in Table VI.

## VII. CONCLUSION

To address the challenge of unpacking Android applications obfuscated by sophisticated packers, we introduced BPFDex, a cutting-edge kernel-based unpacking tool. BPFDex employs eBPF to monitor packing processes and memory data efficiently, significantly minimizing overhead. It successfully unpacks applications protected by various packing mechanisms, discerns anti-unpacker behaviors, and thereby enhances the static analysis of Android malware.

## REFERENCES


[1] Statista.*Android—Statistics & Facts*. [Online]. Available: https://www.statista.com/topics/876/android/#topicOverview

[2] Kaspersky.(2022). *Mobile Threat Report 2022*. Kaspersky Lab. [Online]. Available: https://securelist.com/mobile-threat-report-2022/108844/

[3] (2023). *PHA Categories, Google Developers. Google.* [Online]. Available: https://developers.google.com/android/play-protect/phacategories?

[4] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 56–65.

[5] 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 *Proc. NDSS*, 2012, vol. 25, no. 4, pp. 50–52.

[6] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2011, p. 22.

[7] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 229–240.

[8] Y. Duan et al., "Things you may not know about Android (Un)packers: A systematic study based on whole-system emulation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 18–21.

[9] (2023). *Trustlook, Bangle: Android App Packer Unpacking, Trustlook Blog*. [Online]. Available: https://blog.trustlook.com/bangle-android-app-packer-unpacking/

[10] L. Xue et al., "Happer: Unpacking Android apps via a hardware-assisted approach," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1641–1658.

[11] R. Yu, "Android packers: Facing the challenges, building solutions," in *Proc. 24th Virus Bull. Int. Conf.*, 2014, pp. 266–275.

[12] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed Android applications," in *Proc. 20th Eur. Symp. Res. Comput. Secur. Comput. Secur. (ESORICS)*, 2015, pp. 293–311.

[13] (2023). *QEMU*. [Online]. Available: https://www.qemu.org/

[14] Man7.org.*Ptrace–Linux Manual Page*. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

[15] Frida. (2023). *Frida—A World-Class Dynamic Instrumentation Framework*. GitHub. [Online]. Available: https://github.com/frida/frida

[16] Valgrind. (2023). *Valgrind: An Open-source Memory Debugger*. [Online]. Available: https://valgrind.org/

[17] T. Strazzere. *Android Unpacker: Tools for Unpacking Android APKs*. GitHub. [Online]. Available: https://github.com/strazzere/android-unpacker

[18] L. Xue et al., "PackerGrind: An adaptive unpacking system for Android apps," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 551–570, Feb. 2022.

[19] Halfkiss. (2023). *ZjDroid: Dynamic Java Code Instrumentation Tool for Android*. GitHub. [Online]. Available: https://github.com/halfkiss/ZjDroid

[20] Hex-Rays. (2023). *IDA Pro*. Hex-Rays.[Online]. Available: https://hex-rays.com/IDA-pro/

[21] C. Sun, H. Zhang, S. Qin, N. He, J. Qin, and H. Pan, "DexX: A double layer unpacking framework for android," *IEEE Access*, vol. 6, pp. 61267–61276, 2018.

[22] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "AppSpear: Automating the hidden-code extraction and reassembling of packed Android malware," *J. Syst. Softw.*, vol. 140, pp. 3–16, Jun. 2018.

[23] Z. Ning and F. Zhang, "DexLego: Reassembleable bytecode extraction for aiding static analysis," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 690–701.

[24] H. Zhou et al., "NCScope: hardware-assisted analyzer for native code in Android apps," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 629–641.

[25] eBPF. (2023). *What is EBPF?*. eBPF,.io. [Online]. Available: https://ebpf.io/what-is-ebpf/

[26] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, Jan. 2021.

[27] oyamo.(2023). *Jadx*. GitHub. [Online]. Available: https://github.com/oyamo/jadx

[28] W. Tu, "Bringing the power of eBPF to open vSwitch," in *Proc. Linux Plumbers Conf.*, 2018, p.11.

[29] L. Xue, Y. Yan, and L. Yan, "Parema: An unpacking framework for demystifying VM-based Android packers," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2021, pp. 152–164.

[30] Iovisor.*BCC*. GitHub repository. [Online]. Available: https://github.com/iovisor/bcc

[31] eBPF. (2023). *EBPF Safety*. [Online]. Available: https://ebpf.io/what-is-ebpf/#ebpf-safety

[32] Y. Agman and D. Hendler, "BPFroid: Robust real time Android malware detection framework," 2021, *arXiv:2105.14344*.

[33] CodingGay.(2023). *BlackDex*. GitHub repository. [Online]. Available: https://github.com/CodingGay/BlackDex

[34] WrBug.(2023). *DumpDex*. GitHub Repository. [Online]. Available: https://github.com/WrBug/dumpDex

[35] Hluwa.*Frida-Dexdump*. GitHub. [Online]. Available: https://github.com/hluwa/frida-dexdump

[36] H. Lengyue. *FART*. GitHub. [Online]. Available: https://github.com/hanbinglengyue/FART

[37] U. Youlor. (2023). *Unpacker*. GitHub. [Online]. Available: https://github.com/youlor/unpacker

[38] Android Developers, Google. (2023). *Kernel Security Overview*. [Online]. Available: https://source.android.com/docs/security/overview/kernel-security?hl=zh-cn

[39] Y. Shao, X. Luo, and C. Qian, "RootGuard: Protecting rooted Android phones," *Computer*, vol. 47, no. 6, pp. 32–40, Jun. 2014.

[40] A. Kurmus and R. Zippel, "A tale of two kernels: Towards ending kernel hardening wars with split kernel," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 1366–1377.

[41] F. Mercaldo, C. A. Visaggio, G. Canfora, and A. Cimitile, "Mobile malware detection in the real world," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 744–746.

[42] Android Developers, Google. (2023). *Android NDK*. [Online]. Available: https://developer.android.com/ndk

[43] Android.(2023). *Kernel BPF — Android Open Source Project*. [Online]. Available: https://source.android.com/devices/architecture/kernel/bpf

[44] Kernel. org. (2023). *BPF Verifier*. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/verifier.html

[45] (2023). *PT_REGS_PARM* Macros*. [Online]. Available: arch/arm/include/asm/ptrace.h

[46] Linux Kernel Documentation. (2023). *BPF-Maps*. [Online]. Available: https://www.kernel.org/doc/html/v5.18/bpf/maps.html

[47] PNF Software.(2023). *JEB Decompiler By PNF Software—Android Decompilation*. [Online]. Available: https://www.pnfsoftware.com/jeb/android

[48] L. Li et al., "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017.

[49] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.

[50] T. Ni. (2023). *Eadb: Android Debug Bridge Enhanced*. [Online]. Available: https://github.com/tiann/eadb

[51] J. Schütte, R. Fedler, and D. Titze, "ConDroid: Targeted dynamic analysis of Android applications," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2015, pp. 571–578.

[52] J. Freke. (2023). *Smali*. GitHub. [Online]. Available: https://github.com/JesusFreke/smali

[53] rovo89. (2023). *Xposed*. GitHub. [Online]. Available: https://github.com/rovo89/Xposed

[54] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, Jun. 2014, pp. 447–458.

[55] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 33–49.

[56] F-Droid. (2023). *F-Droid—Free and Open Source Android App Repository*. [Online]. Available: https://f-droid.org/

[57] Alibaba Cloud.(2023). *Mobile PaaS—Public Products and Solutions*. Alibaba Cloud. [Online]. Available: https://www.aliyun.com/product/mobilepaas/mpaas_ppm_public_cn

[58] Baidu.(2023). *APK Protect*. [Online]. Available: https://apkprotect.baidu.com/

[59] Bangcle.(2023). *Homepage*. [Online]. Available: https://dev.bangcle.com/

[60] IJiami.(2023). *IJiami Security Solutions—Android*. [Online]. Available: https://www.ijiami.cn/android

[61] 360 JiaGu. (2023). *360 JiaGu*. [Online]. Available: https://jiagu.360.cn/#/global/index

[62] Tencent Cloud, Tencent Holdings Limited. (2023). *Microservices (MS)—Product Overview*. [Online]. Available: https://cloud.tencent.com/product/ms

[63] KiwiSec.(2023). *Kiwi Security*. [Online]. Available: https://www.kiwisec.com/

[64] Nagain.(2023). *Nagain Home*. [Online]. Available: https://www.nagain.com/#/home/index

[65] Testin. cn. (2023). *Testin*. [Online]. Available: https://www.testin.cn/

[66] SourceForge.(2023). *APKProtect Project*. [Online]. Available: https://sourceforge.net/projects/apkprotect/

[67] Massachusetts Institute of Technology. (2023).*Tools for Debugging*. [Online]. Available: https://stuff.mit.edu/afs/sipb/project/android/docs/tools/debugging/index.html

[68] S. Arzt et al., "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[69] V. Avdiienko et al., "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 426–436.

[70] C. Qian, X. Luo, Y. Le, and G. Gu, "VulHunter: Toward discovering vulnerabilities in Android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan. 2015.

[71] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surveys*, vol. 49, no. 4, pp. 1–41, Dec. 2017.

[72] M. Xu et al., "Toward engineering a secure Android ecosystem: A survey of existing techniques," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 1–47, Jun. 2017.

[73] Primate Labs.(2023). *Geekbench—Cross-Platform Benchmark*. [Online]. Available: https://www.geekbench.com/

[74] Android Developers, Google. (2023). *UI/Application Exerciser Monkey*. [Online]. Available: https://developer.android.google.cn/studio/test/monkey

[75] J. Cai, T. Li, C. Huang, and X. Han, "ReDex: Unpacking Android packed apps by executing every method," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 337–344.

[76] H. Lu, C. Jin, X. Helu, X. Du, M. Guizani, and Z. Tian, "DeepAutoD: Research on distributed machine learning oriented scalable mobile communication security unpacking system," *IEEE Trans. Netw. Sci. Eng.*, vol. 9, no. 4, pp. 2052–2065, Jul. 2022.