Happer: Unpacking Android Apps via a Hardware-Assisted Approach

Lei Xue[1*], Hao Zhou[1*], Xiapu Luo[1§], Yajin Zhou[2], Yang Shi[3], Guofei Gu[4], Fengwei Zhang[5], Man Ho Au[6]
[1]The Hong Kong Polytechnic University, [2]Zhejiang University
[3]Tongji University, [4]Texas A&M University
[5]Southern University of Science and Technology, [6]The University of Hong Kong

***Abstract*—Malware authors are abusing packers (or runtime-based obfuscators) to protect malicious apps from being analyzed. Although many unpacking tools have been proposed, they can be easily impeded by the anti-analysis methods adopted by the packers, and they fail to effectively collect the hidden Dex data due to the evolving protection strategies of packers. Consequently, many packing behaviors are unknown to analysts and packed malware can circumvent the inspection. To fill the gap, in this paper, we propose a novel hardware-assisted approach that first monitors the packing behaviors and then selects the proper approach to unpack the packed apps. Moreover, we develop a prototype named `Happer` with a domain-specific language named behavior description language (BDL) for the ease of extending `Happer` after tackling several technical challenges. We conduct extensive experiments with 12 commercial Android packers and more than $24k$ Android apps to evaluate `Happer`. The results show that `Happer` observed 27 packing behaviors, 17 of which have not been elaborated by previous studies. Based on the observed packing behaviors, `Happer` adopted proper approaches to collect all the hidden Dex data and assembled them to valid Dex files.**

I. Introduction

Malware authors have been abusing packers (or runtime-based obfuscators [70]) to protect malicious apps from being analyzed [6]. We call the apps protected by packers as packed apps and their protection behaviors as packing behaviors, such as detecting the running environment, hiding the original Dex data (i.e., the Dex file and the Dex items that constitute the Dex file) and then releasing them at runtime, to name a few. Many unpacking tools (i.e., unpackers) for Android apps [3, 7, 8, 36, 45, 56, 70, 72, 79, 86] have been developed to recover the packed app's original Dex file. They retrieve the *hidden* Dex data that has been released to the memory through a customized Android system [56, 70, 79, 86], or a debugger [3], or a dynamic binary instrumentation (DBI) framework [36, 72], or a full Android system emulator [7, 8, 45].

Since arms race between packers and unpackers never ends, packers are evolving to render unpackers ineffective. In particular, they adopt various approaches to detect the presence of unpackers and impede the analysis as well as change the write-then-execute pattern to prevent unpackers from collecting all the hidden Dex data [1, 2]. To deal with the ever-evolving packers, we argue that a long-lasting effective unpacker should first scrutinize the packing behaviors of the packed apps and then choose the proper unpacking strategies accordingly, instead of using the fixed unpacking patterns. In particular, it should fulfill the following three requirements.

- **`R1`-Effectiveness**: it should retrieve all the hidden Dex data and assemble them into a valid Dex file. Effectiveness is the basic requirement for unpackers;
- **`R2`-Extensibility**: it should be easily extended to cover new packing behaviors;
- **`R3`-Adaptivity**: it should select proper unpacking strategies according to the observed packing behaviors of packed apps.

Unfortunately, none of existing unpackers satisfies these requirements. *First*, they do not fulfil **`R1`**, because they [7, 36, 72] rely on debuggers, emulators, or DBI frameworks and thus can be easily detected by packers due to their noticeable fingerprints, such as special strings left in the memory. Once detected, packers may stop releasing the hidden Dex data. Meanwhile, packers usually hook system library methods (e.g., *write* and *__android_log_buf_write*) to impede unpackers [8, 53, 56, 70, 79, 86] that use such methods to dump the released Dex data. Moreover, as shown in Table I, existing unpackers fail to obtain all the hidden Dex data. Specifically, some unpackers [7, 8, 36, 86] will obtain useless Dex files that contain invalid Dex items. Others [53, 56, 70, 72, 79] can just retrieve partial Dex items because packers may release the real code right before execution and then delete it. *Second*, existing unpackers are usually hard to be extended because they rarely consider the evolution of packers, and thus do not meet **`R2`**. *Third*, almost all existing unpackers use fixed unpacking strategies and thus can be easily evaded and do not fulfill **`R3`**.

In light of these limitations, we propose a novel hardware-assisted approach and develop a new tool named `Happer`, which fulfills the three requirements, to revisit the latest Android packers and reveal the packing behaviors missed by existing unpackers. **`R1`**: `Happer` tracks the packed app's runtime behav-

TABLE I: An overview of existing unpackers in academia.

Unpacker	Effectiveness	Hidden Dex file[1]	Hidden Dex item[1]	#Behaviors[2]
DexHunter-15 [86]	✗	●	⊗	**3**
AppSpear-15 [79]	✗	●	◐	**4**
PackerGrind-17 [72]	✗	●	◐	**7**
AppSpear-18 [53]	✗	●	◐	**6**
DroidUnpack-18 [45]	✗	●	◐	**3**
DexLego-18 [56]	✗	●	◐	**N/A**
TIRO-18 [70]	✗	●	◐	**4**
Happer	✓	●	●	**27**

[1] ●, ◐, and ⊗ indicate that all of, partial of, and none of the hidden Dex file (or the other hidden Dex data), respectively, which will be dynamically released by the packers, can be collected by the unpacker.
[2] #Behavior denotes the number of supported packing behaviors (more details in Table IV).

* Co-first authors.
§ The corresponding author.

iors and dumps the hidden Dex data by leveraging the hardware features of the ARM platform, including ETM (Embedded Trace Macrocell) and HBRKs (Hardware Breakpoints), so that it cannot be easily impeded by packers. Meanwhile, we let the packed app release all the hidden Dex data at runtime, and thus `Happer` can collect them to effectively recover the original Dex file. **`R2`**: We design a domain-specific language BDL for analysts so that they can easily specify the (new) packing behaviors to be tracked by `Happer`. Currently, `Happer` supports 27 packing behaviors, much more than the number of packing behaviors reported by previous studies. **`R3`**: Instead of adopting a fixed strategy to unpack apps, `Happer` selects the proper Dex data collection points according to the observed packing behaviors.

More specifically, to expose the runtime packing behaviors and unpack the packed app, we leverage ETM [17] to track the executed instructions of the packed app, and then analyze the execution trace to recover the runtime method invocations performed by the packed app. Meanwhile, we use HBRKs to obtain the concrete parameter values of system functions at runtime. After that, we determine the packing behaviors based on the tracked information and select the proper Dex data collection points for collecting the dynamically released Dex data. Finally, we assemble the retrieved Dex data into a valid Dex file that can be analyzed by the off-the-shelf static analysis tools. To realize the whole process, we tackle three major technical challenges. **C1**: the content (e.g., the dynamically released Dex data) mapped in virtual memory are not loaded into physical memory synchronously. To address **C1**, we design a method to force the OS to load the virtually mapped pages into physical memory. **C2**: there is a semantic gap between the tracked instruction addresses and the high-level packing behaviors. To handle **C2**, we build a mapping between system functions (including system library methods and framework APIs) and binary instructions in advance by resolving the loaded system libraries and Android framework's Oat files. The mapping facilitates the recovery of the system functions that has been invoked by the packed app. **C3**: advanced packers release the hidden Dex data just before the data will be used by the Android runtime, which prevents unpackers from retrieving all the hidden data. To tackle **C3**, we propose an approach to force the packed app to release all the hidden Dex data to virtual memory at runtime.

We conduct extensive experiments with 12 commercial Android packers and more than $24k$ Android apps to evaluate `Happer`. The results show that `Happer` observed 27 packing behaviors, 17 of which have not been elaborated by previous studies. Based on the observed packing behaviors, `Happer` adopted proper approaches to collect all the hidden Dex data and assembled them to valid Dex files.

In summary, we make the following contributions:

- We propose the *first* hardware-assisted approach to scrutinizing the packing behaviors and then unpacking the packed app accordingly. Our approach is effective, extensible and adaptive to the ever-evolving packers.
- We develop a new unpacking tool named `Happer` after tackling multiple technical challenges, and it is released at https://github.com/rewhy/happer.
- We perform extensive experiments using 12 commercial packers and more than $24k$ apps. `Happer` observed 27 packing behaviors, 17 of which have not been thoroughly reported. Besides, `Happer` can retrieve all hidden Dex data of the packed apps and resemble them to valid Dex files.

II. Background

A. Hardware Features of ARM Platform

The ARM architecture defines non-invasive and invasive tracing features to perform the tracing tasks [11]. The non-invasive tracing components monitor the target process whereas the invasive tracing components not only monitor the target process but also control it. In this paper, we exploit the non-invasive tracking component ETM and the invasive tracking component HBRKs to conduct dynamic tracking.

ETM traces the instructions and data by monitoring the instruction and data buses directly with little additional overhead [17]. During tracing, ETM generates *trace element*s containing the instruction and data information for recovering the program's runtime execution traces. There are 30 types of *trace element*s, and they are produced in the form of stream and stored in a 64 KB on-chip buffer or an up to 4 GB external device (i.e., `DSTREAM`). We focus on resolving the *Address* and the *Context* elements from the stream, because they carry the tracked instruction's virtual address and the traced program's context identifier (i.e., PID), which are used to recover the runtime behaviors of the tracked app (§V-A) and filter out irrelevant trace elements (§IV-A), respectively.

HBRKs are proposed to interrupt the process during tracing. When a HBRK is hit, a debugging event will rise and then the debugger/tracker can handle this event and let the processor enter debugging mode. Different from the software breakpoints (SBRKs) that need to replace the original instructions with `brk` instructions, HBRKs do not change original instructions, and thus HBRKs are harder to be noticed.

B. A Motivating Example

Fig. 1 presents the code snippets of a motivating example for `Happer`, which is summarized according to the packing behaviors of real commercial packers. The example consists of a simplified static initialization code for an app class (Fig. 1a) and a representative implementation of a native library of the app, which is added by the packer (Fig. 1b).

As shown in Fig. 1a, the packer inserts the declarations of 4 JNI methods (Line 1-4) and a static initialization code (Line 6-11) to the packed app. Meanwhile, the original bytecode of the *onCreate* callback (Line 5) is replaced with the stub code, i.e., the `return-void` instruction. To ensure that the inserted JNI methods (i.e., the behaviors of packers) are executed before the protected bytecode, invocations to these methods are placed in the static initialization method, because such method will always be executed before any other code in this class.

```
1 public native void selfCheck();
2 public native void detectEnvironment();
3 public native void preventUnpacking();
4 public native void recoverDexData();
5 protected void onCreate(Bundle bundle) {} // "return-void"
// The static initialization method declared in the Application class
// Or the attachBaseContext method defined in the Application class
6 static {
7     selfCheck();          // Check the integrity of the protected Dex file
8     detectEnvironment(); // Detect the running environment
9     preventUnpacking();  // Prevent the collection of the related Dex data
10    recoverDexData();    // Recover the bytecode of the onCreate method
11 }
```

(a) The implementation of the protected Dex file in the packed app.

```
1 void selfCheck() {          // Detect the presence of brk instruction
2     if (detectBrkInstruction())
3         kill();              // Kill the process
4 }
5 void detectEnvironment() { // Examine the running environment
6     if (isDebugger() || isEmulator() || isDBIFramework())
7         kill();              // Terminate the process
8 }
9 void preventUnpacking() {  // Prevent the Dex data from being collected
10    hookLibcFunction();    // Hook the system methods declared in libc.so
11    hookLiblogFunction();  // Hook the system methods declared in liblog.so
12 }
13 void recoverDexData() {    // Recover the Dex data of the packed Dex file
14    adjustDexData();       // Override or readdress the protected Dex data
15 }
```

(b) The implementation of the native library in the packed app.

Fig. 1: A motivating packed app.

The implementations of the inserted JNI methods are shown in Fig. 1b. *selfCheck* detects the presence of `brk` in virtual memory (Line 1-4). Once found, the packed app will terminate itself. Otherwise, *detectEnvironment* examines whether the packed app is running in an emulator, or a debugger, or a DBI framework (Line 5-8). If the fingerprint of a particular running environment is detected, the packed app will intentionally kill itself. Moreover, to prevent the released Dex data from being dumped via the methods provided by the system libraries (e.g., *write* defined in `libc.so` and *__android_log_buf_write* defined in `liblog.so`), *preventUnpacking* hooks them (Line 9-12) to disable their functionality or inspect whether they are called to access the released Dex data. If all these checks are passed, the packer calls *recoverDexData* to release the hidden Dex data (Line 13-15) (i.e., the original bytecode of *onCreate*).

Existing unpackers (e.g., ZjDroid [36], and PackerGrind [72]), which are implemented upon debuggers, emulators, or DBI frameworks, fail to unpack this example, because they cannot pass the check of the running environment. Similarly, the unpackers (e.g., DexHunter [86], DexLego [56], TIRO [70], AppSpear [53, 79], and android-unpacker [7]), which dump the released Dex data by using system library methods (e.g., *write*), cannot unpack this example, because it hooks these methods to prevent itself from being unpacked.

To overcome the limitations of existing unpackers, we design and implement a novel hardware-assisted unpacker, named `Happer`. It can circumvent all the aforementioned checks to track the packer's behaviors and retrieve the hidden Dex data by leveraging the hardware features of the ARM platform. `Happer` also assembles the collected Dex data to valid Dex files as the unpacking results, which can be fed to existing app analysis tools (e.g., `FlowDroid` [39]).

III. High-level System Design

In this section, we introduce the overview (§III-A) and the workflow (§III-B) of `Happer`. Then, we discuss the encountered technical challenges and the corresponding solutions (§III-C).

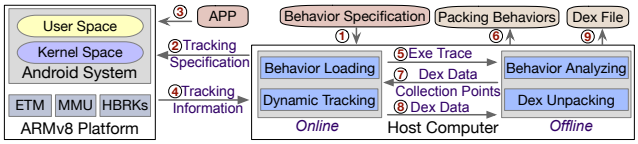


Fig. 2: The overview and workflow of Happer.

A. Overview

As shown in Fig. 2, `Happer` consists of an ARMv8 platform for conducting the dynamic tracking and a host computer for controlling the tracing task, analyzing the retrieved information, and assembling the collected Dex data. `Happer` takes in the packed app and the behavior specifications that instruct `Happer` to track the packing behaviors. Then, `Happer` will output the observed packing behaviors and the reassembled Dex file. Note that, in order to extend the types of packing behaviors supported by `Happer`, we provide a domain-specific language (§V-B) for analysts to specify the packing behaviors to be tracked.

`Happer` requires no modification to the Android system and the packed app under analysis, because it uses the hardware features to track the packing behaviors and unpack the app. To minimize the additional overhead introduced to the execution of the packed app, the behavior specifications for setting up the tracing tasks and the dynamically retrieved data will be processed on the host computer rather than the ARM platform. More specifically, the host computer consists of an online and an offline component. The online component loads and parses the input behavior specifications and then instructs the ARM platform to conduct the real-time tracking and retrieve the runtime data (e.g., the execution trace, the released Dex data, and the demanded memory content). Since analyzing packing behaviors and assembling Dex data are time-consuming tasks, they are carried out by the offline component.

Assumption: We assume that packers cannot gain the root privilege to load a kernel module that detects the presence of `Happer` by inspecting the values of the registers for configuring ETM. This assumption is rational because Google has adopted various strategies to protect the kernel of Android system [31], and smartphone vendors also use the techniques, such as SPLIT KERNEL [52] and RootGuard [64], to further harden the kernels of their customized Android systems.

B. Workflow

Given a packed app, `Happer` analyzes it through three phases.

(1) Monitoring Runtime Behaviors (§IV). In this phase, `Happer` loads and parses the behavior specifications (Step ①), and then instructs the ARM platform to launch the packed app (Step ②) and conduct the tracing task (Step ③). Meanwhile, `Happer` retrieves the necessary runtime data for determining packing behaviors, including the ETM stream, the demanded memory content, and the released Dex data (Step ④).

(2) Analyzing Packing Behaviors (§V). In this phase, `Happer` parses the ETM stream to recover the runtime method invocations conducted by the packed app (Step ⑤). Then, `Happer`

determines the packing behaviors according to the resolved method invocations, the retrieved memory content, and the dumped Dex data (Step ⑥).

(3) Unpacking Packed Apps (§VI). In this phase, `Happer` retrieves the dynamically released Dex data and then assembles them to a Dex file as the unpacking result. More specifically, according to the recognized packing behaviors, `Happer` selects the proper Dex data collection points (Step ⑦) and instructs the online component to dump the relevant Dex data (Step ⑧). Then, the offline component assembles the retrieved Dex data to a valid Dex file (Step ⑨).

C. Challenges and Solutions

Developing `Happer` based on the hardware features of ARM platform needs to address the following three challenges.

C1: Fetching Memory Data. Unlike self-hosted debuggers that run along with the target process and fetch the memory data from virtual memory, `Happer` works as an external debugger and retrieves data from physical memory. However, not all the data mapped in virtual memory is loaded into physical memory by the OS synchronously. For example, if a process uses *mmap* to load a file into virtual memory, its content is just actually loaded into physical memory when being accessed. If the data accessed by a process is not in physical memory, a page fault will be triggered, and then the kernel will load the required data into physical memory.

Solution: We propose a DBI-based approach, which will insert additional instructions to the memory and make OS execute the instructions to completely load the demanded data (e.g., the dynamically released Dex file) into physical memory before `Happer` retrieves the data. More details are presented in §IV-B.

C2: Resolving ETM Stream. Existing ETM stream resolvers (e.g., `ptm2human` [10] and `ds-5` [9]) focus on parsing the virtual addresses of the tracked instructions. However, detecting packing behaviors also requires high-level semantic information, such as framework and native method invocations. Hence, there is a semantic gap between the packing behaviors and the tracked information resolved by the existing resolvers.

Solution: To bridge this semantic gap, we design a new resolver to resolve instruction-level information from ETM stream and also map them to the native level and framework method level information by parsing the information from the loaded system libraries and Oat files [20].

C3: Collecting Dex data. Advanced packers (e.g., Ijiami [19] and Kiwi [24]) release the hidden Dex data just before it is used by Android runtime. Since existing unpackers cannot let these packers release all hidden Dex data during unpacking, the recovered Dex files do not contain all hidden Dex data.

Solution: To force the packed app to release all the hidden Dex data, we propose a hook-based approach, which will adjust the execution flows of the Android runtime by modifying the memory data and the register values. Specifically, according to the identified packing behaviors of the target app, we hook specific system library methods that are called by the Android runtime to resolve the Dex data. More details about collecting the Dex data are elaborated in §VI-B.

IV. Monitoring Runtime Behaviors

This section elaborates how `Happer` monitors the packing behaviors of the packed apps, including tracking the execution flows (§IV-A) and fetching the memory data (§IV-B).

A. Tracing Execution Flow

`Happer` uses ETM, especially the context ID tracing and the branch broadcasting tracing functionality, to track the execution flows of the packed app. Note that, to tackle the issue that the capacity of the on-chip buffer (64KB) is insufficient for storing the generated ETM stream, we use `DSTREAM`, a dedicated off-chip device with 4GB buffer, to store the ETM stream.

(1) Context ID Tracing: To monitor the packed app's packing behaviors, `Happer` needs to track the app's runtime execution flows. Note that, by default, ETM will record the target address of each indirect branch instruction executed by CPU. To filter out the irrelevant trace elements, `Happer` enables and configures the context ID tracing of ETM to make it only trace the process that corresponds to the packed app.

Specifically, `Happer` sets the `CID` field of `TRCCONFIGR` to 0x1. Then, `Happer` writes the packed app's PID to the `TRCCIDCVR0` register and changes the `COMP0` field of `TRCCIDCCTLR0` to 0x1, which makes ETM only track the execution flows of a specific process according to the PID value stored in `TRCCIDCVR0`.

(2) Branch Broadcast Tracing: By default, ETM only records the target address of each executed indirect branch instruction. However, to recover the runtime method invocations of the packed app, the target addresses of both the direct and indirect branch instructions are needed, because both of them can lead to the method invocations. To this end, `Happer` enables the branch broadcast tracking of ETM to make it also record the target address of each executed direct branch instruction.

Specifically, `Happer` enables the branch broadcast tracing by setting the `BB` field of `TRCCONFIGR` to 0x1. Then, to specify the memory region for applying the branch broadcast tracing, `Happer` writes 0x0 and 0xffffffff to `TRCACVR0` and `TRCACVR1`, respectively, which indicates that the branch broadcast tracing will be conducted within the entire virtual memory space of the packed app. Subsequently, we set the `RANGE` field of `TRCBBCTLR` to 0x1, which instructs ETM to also record the target address of each executed direct branch instruction.

B. Fetching Memory Data

Working as an external debugger, `Happer` cannot access virtual memory of the packed app, because the packed app does not share the same context with `Happer`. To retrieve the data in the packed app's virtual memory, `Happer` takes 2 steps. First, `Happer` calculates the physical address of the demanded data according to its virtual address. Second, `Happer` instructs the OS to completely load the demanded data into physical memory before `Happer` retrieves the data. We elaborate more on these 2 steps as follows. The MMU registers used for fetching the memory data are summarized in Appendix-A.

```
   input : addr is the start address of the target virtual memory space;
           size is the size of the target virtual memory space.
1  Function force_loading(addr, size):
2      if require_force_loading(addr, size) == false then
3          VA_code = code_mem_determine()
4          PC_exe = backup_execution_environment(VA_code)
5          do_force_loading_wrapper(VA_code, addr, size)
6          restore_execution_environment(PC_exe)
7      end
8  return
```

Fig. 3: Loading unmapped data into physical memory.

(1) Calculating Physical Memory Address: Since the traced process, corresponding to the packed app, manipulates the data in virtual memory, Happer transfers the virtual memory address to its corresponding physical memory address in order to fetch the target data. In practice, Happer uses MMU to achieve this purpose and more details are presented in Appendix-B.

(2) Loading Memory Data: To address the challenging issue that the data mapped in virtual memory may not be loaded into physical memory synchronously (**C1**), we propose an approach to force the OS to completely load the data into physical memory before Happer fetches it. Our approach consists of 2 steps as shown in Fig.3.

In the first step, referring to *require_force_loading* at Line 2, Happer checks whether the data has already been completely loaded into physical memory. In particular, since the data (e.g., the hidden Dex file) is loaded into virtual memory with the page granularity [38], we split the memory region, storing the data, into several memory slices according to the page size. Then, we calculate the physical address of each slice by using MMU in order to check whether the data has been completely loaded into physical memory. If MMU fails to calculate the physical address of a particular slice, namely the F field of PAR_EL1 is 0x1, the data has not been entirely loaded. Thus, Happer will force the OS to load the target data.

In the second step (in Line 3-6), Happer forces the OS to load the data into physical memory. We first insert instructions for loading the target data to a particular memory region with the rwx permission (i.e., readable, writable, and executable), and then we instruct CPU to execute these instructions.

However, it is non-trivial to find a suitable memory region because the executable permission and the writable permission are rarely enabled together due to the security consideration [31]. Instead, we enable the writable permission of an executable memory region. Specifically, in *code_mem_determine* at Line 3, Happer obtains the physical address of the translation table[1] according to the BADDR field of TTBR0_EL1. Then, Happer finds a page descriptor with its UXN field set to 0x1, which indicates that the corresponding memory region is executable. Note that, before Happer sets the AP field of the found page descriptor to 0x01, which makes the associated memory region readable and writable, in *backup_execution_environment* at Line 4, we back up the context information, including the page descriptor value, the content of the corresponding memory region, and the values of PC, R0, and R1 registers, because this context information will then be modified. Meanwhile, we also store the virtual address of the found memory region to VA_{code}. After that, we write the instructions for loading the data to the found memory region and set the PC register to VA_{code} so that the OS will execute the instructions (i.e., *do_force_loading_wrapper* at Line 5). Finally, in *restore_execution_environment*, we recover the original context information and resume the execution. The inserted loading instructions are presented in Appendix-C.

[1]The translation table is placed in physical memory, and it defines the properties of different memory regions with various sizes from 1KB to 1MB.

```
   input : address is the address of an instruction recorded by ETM;
           pre_address is the address of the last handled instruction;
           map is the mapping from instruction address to system method.
   output: call, the cross-layer method call graph;
1  Function identify_normal_invocation(address, pre_address, map):
2      if address is the start address of a system method then
3          method = map[address]
4          if pre_address is an instruction of the target app then
5              pre_invoke = method
6              call.add(<method, NORM>) // Invocation to library function.
7          end
8          if stack.peek(1)==quick_trampoline and stack.peek(2)==APP* then
9              call.add(<method, NORM>) // Invocation to Oat method.
10         end
11         if pre_address == invoke_stub and pre_invoke == JNI_CALL then
12             call.add(<method, REFL>) // JNI reflection invocation.
13         end
14     end
15 return
```

* APP presented in the algorithm refers to the method defined in the target app.

Fig. 4: Identifying invocations to system library methods.

V. Analyzing Packing Behaviors

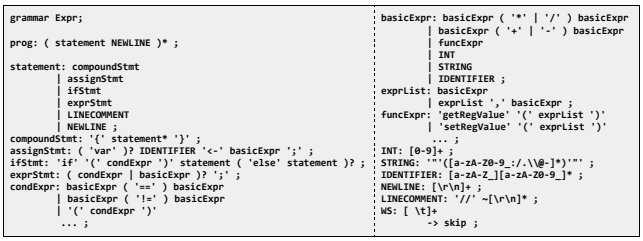
This section details how Happer finds the packing behaviors according to the tracked information, including resolving the ETM stream (in §V-A) and identifying the packing behaviors (in §V-C). Meanwhile, in §V-B, we introduce the domain-specific language for specifying the packing behaviors.

A. Resolving ETM Stream

To identify the packing behaviors, Happer needs to get the system functions (i.e., framework APIs and system library methods) that have been invoked by the packed app at runtime from the ETM stream. However, none of existing ETM stream resolvers can achieve this purpose (**C2**). Therefore, we develop a new resolver that maps instruction addresses to system functions in advance and then recognizes the invoked system functions according to the addresses of the tracked instructions.

(1) Constructing Address-Function Mapping: To assist the identification of the invoked system functions, we construct a mapping between instruction addresses and system functions by analyzing the memory map of the packed app and the disassembled information extracted from system libraries and Oat files. Since constructing the address-function mapping is a common practice for dynamic app analysis tools [73, 77, 78], we leave the details of our process in Appendix-D.

(2) Resolving Method Invocation: Leveraging the constructed address-function mapping, Happer determines the system functions invoked by the packed app through resolving the addresses of the tracked instructions as shown in Fig.4. Specifically, in Line 2, we examine whether the address of a tracked instruction (i.e., the input *address*) is the start address of a system function. If so, an invocation to the system function is found. Since a



```
grammar Expr;

prog: ( statement NEWLINE )* ;

statement: compoundStmt
        | assignStmt
        | ifStmt
        | exprStmt
        | LINECOMMENT
        | NEWLINE ;
compoundStmt: '{' statement* '}' ;
assignStmt: ( 'var' )? IDENTIFIER '<-' basicExpr ';' ;
ifStmt: 'if' '(' condExpr ')' statement ( 'else' statement )? ;
exprStmt: ( condExpr | basicExpr )? ';' ;
condExpr: basicExpr ( '==' ) basicExpr
        | basicExpr ( '!=' ) basicExpr
        | '(' condExpr ')'
        ... ;

basicExpr: basicExpr ( '*' | '/' ) basicExpr
        | basicExpr ( '+' | '-' ) basicExpr
        | funcExpr
        | INT
        | STRING
        | IDENTIFIER ;
exprList: basicExpr
        | exprList ',' basicExpr ;
funcExpr: 'getRegValue' '(' exprList ')'
        | 'setRegValue' '(' exprList ')'
        ... ;
INT: [0-9]+ ;
STRING: '"'([a-zA-Z0-9_:/.\\@-]*)'"' ;
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]* ;
NEWLINE: [\r\n]+ ;
LINECOMMENT: '//' ~[\r\n]* ;
WS: [ \t]+
        -> skip ;
```

Fig. 5: The core grammar specification for BDL.

system function can be invoked by either the packed app or other system functions, we further determine whether or not the system function is called by the packed app in Line 4-13 to be explained as follows.

Since the packed app can adopt three ways to invoke a system function, we design different mechanisms to handle them. First, Line 4-6 handles the invocation to the system library method. If the tracked instruction presents just before the execution of a system library method and the instruction is located in the code segment of the packed app, a system library method is found to be invoked by the packed app. Second, Line 8-10 handles the invocation to the framework API. If the packed app's code (i.e., APP* at Line 8) calls the ART runtime function *art_quick_resolution_trampoline* (i.e., *quick_trampoline* at Line 8), a framework API is found to be invoked by the packed app. Third, Line 11-13 handles the case that the framework API is invoked by JNI reflection. If there is an invocation from the function *JNI::Callxxxx* or *JNI::NewObjectxxxx* (i.e., *JNI_CALL* at Line 11) to *art_quick_invoke_stub_internal* (i.e., *invoke_stub* at Line 11), a framework API is found to be invoked by the packed app through JNI reflection.

Since advanced packers may hook system library methods to protect the released Dex data from being dumped by unpackers, we determine such packing behaviors by identifying the hooked system library methods that were invoked by the packed app. More details are presented in Appendix-E.

B. Behavior Description Language (BDL)

The packers evolve frequently to evade unpacking and correspondingly the unpackers should have high scalability to handle the newly adopted packing behaviors. Consequently, inspired by the domain-specific language [50, 51], we propose a specific packing behavior description language (i.e., BDL) based on ANTLR [37], which simplifies the extension of Happer to monitor new packing behaviors and does not require the analysts know the implementation details of Happer.

Fig. 5 shows the core grammar specification for BDL, including two major types of syntax, namely BDL statement and BDL expression. A behavior specification is composed of a sequence of one or more BDL statements, each of which specifies a BDL operation, such as assigning a value to a BDL variable or invoking a BDL function. More specifically, there are four types of BDL statements, including expression-statement (*exprStmt*), compound-statement (*compoundStmt*), if-statement (*ifStmt*), and assignment-statement (*assignStmt*), which are used to specify a common operation (e.g., invoking a BDL function), a block of statements describing a complex operation (e.g., assigning a value to a variable and then passing the variable to a BDL function), a conditional operation, and a variable assignment operation, respectively.

TABLE II: A summary of packing behaviors.

Type	Description	Behaviors
ADG	Checking fingerprints of debuggers.	ADG-1/2/3/4/5/6
AEU	Checking fingerprints of emulators.	AEU-1/2/3
ADI	Checking fingerprints of DBI frameworks.	ADI-1
TCK	Checking time delays incurred by dynamic analysis tools.	TCK-1/2
SLH	Hooking debug/unpack related system library methods.	SLH-1/2/3
DDL	Loading protected Dex files at runtime.	DDL-1/2/3
DDM	Modifying content of protected Dex files.	DDM-1/2/3/4
DOM	Modifying relevant ART runtime objects.	DOM-1/2
DDF	Dispersedly loading content of protected Dex files.	DDF-1/2
JNT	Translating app's function to the native code.	JNT-1

BDL can be employed to extend Happer to monitor any packing behaviors composed of one or more BDL operations. Meanwhile, analysts can configure Happer to just monitor their interested behaviors using BDL for improving efficiency. Moreover, we have used BDL to define the packing behaviors presented in this paper. More details of BDL are presented in Appendix-F with an example.

C. Identifying Packing Behavior

According to the behavior specifications, Happer identifies packing behaviors based on the tracked information. As summarized in Table II, Happer currently supports 10 categories of packing behaviors, including 27 distinct behaviors in total, which are identified after inspecting more than $24k$ apps and 12 commercial packers. We will include more behavior specifications once new packing behaviors are found in future work. Moreover, by releasing Happer, we encourage users to contribute more behavior specifications. In this section, we introduce the implementation details of the identified packing behaviors as well as how Happer detects them.

(1) Anti-Debugging (ADG): Packers use ADG to prevent them from being analyzed by debuggers. Commonly, there are 6 ways for packers to conduct ADG, which could be divided into 4 groups. ① Packers invoke *Debug.isDebuggerConnected* to detect the presence of debuggers (ADG-1). ② Packers call *fork* and *ptrace* to attach the forked process to the packed app, which can prevent the packed app from being attached by debuggers (ADG-2), because a process can be attached by only one another process at a time. ③ Packers examine the process status of the packed app to check whether it is traced by debuggers (ADG-3). ④ Packers hook the system functions related to debugging (i.e., *Instrumentation::AddListener*, *Instrumentation::EnableMethodTracing*, *__android_log_buf_write*) to prevent them from being used by debuggers (ADG-4/5/6).

Detection: ① For ADG-1, Happer inspects the resolved method invocations to check whether *isDebuggerConnected* has been called. ② For ADG-2, Happer examines the method invocations to determine whether *fork* has been invoked. If so, Happer

accesses the process status files (i.e., */proc/pid/status* and */proc/self/status*) to check whether the process of the packed app has been attached. ③ For ADG-3, Happer sets HBRKs to system library methods, *fopen*, *fgets*, and *strncmp*, and then checks their runtime parameters to determine whether the packer searches the "TracerPid" string in the process status file to know it has been traced by debuggers. ④ For ADG-4/5/6, Happer reviews the resolved method invocations to check whether the debugging-related system functions have been hooked.

(2) Anti-Emulator (AEU): Packers prevent themselves from running in emulators (or VMs) because most dynamic analysis frameworks [46, 58, 67, 74, 75, 78] are implemented based on the virtual machine introspection (VMI) [47]. Packers usually detect the presence of emulators via inspecting specific system files (e.g., */proc/tty/drivers*, which contains the name of the registered virtual TTY driver for Qemu) (AEU-1), or checking the existence or values of particular system properties (e.g., *init.svc.qemud*, which is introduced by Qemu) (AEU-2/3).

Detection: For AEU-1, Happer sets HBRKs to *fopen*, *fscanf*, and *strncmp*, and examines their parameters to determine whether the packer inspects the content of specific system files to detect the emulator. Similarly, to identify AEU-2/3, Happer sets HBRKs to *__system_property_get* and *strncmp*, and examines their parameters to determine whether the packer inspects the values of particular system properties to detect the emulator.

(3) Anti-DBI (ADI) Packers detect the presence of DBI frameworks to avoid being instrumented and analyzed. Commonly, packers access the file */proc/pid/maps* and examine the memory map of their processes to find whether the files associated with DBI frameworks are presented in the memory (ADI-1).

Detection: Happer sets HBRKs to *fopen*, *fgets*, and *strstr*, in order to obtain their runtime parameters, and then examines the retrieved values to identify ADI-1.

(4) Time Checking (TCK): Dynamic app analysis introduces additional delays to the execution of the packed app, and thus packers infer whether they are being analyzed via calculating the time consumed for executing a special task [69]. Commonly, packers get the current time using the system library methods, such as *gettimeofday* (TCK-1) and *time* (TCK-2).

Detection: For TCK-1, Happer sets a HBRK to *gettimeofday* and retrieves the pointer that refers to the timeval structure (i.e., the first parameter of *gettimeofday*). Then, Happer sets another HBRK to the return address of *gettimeofday* and obtains the value stored in the timeval structure. If it is not the first time for the packed app to invoke *gettimeofday*, we add an extra interval (more than one second in practice) to the obtained time value and then replace the origin with the adjusted one, which imitates the dynamic app analysis to add additional time delays to the execution of the packed app. If the app crashes or terminates unexpectedly, TCK-1 is identified. Similarly, Happer follows the same steps to recognize TCK-2.

(5) System Library Hooking (SLH): Unpackers can use the system library methods (e.g., *ptrace*, *open*, and *write*) to analyze the packed app or dump the released Dex data. Accordingly, packers may hook the related methods in system libraries, libart.so (SLH-1), libc.so (SLH-2), or liblog.so (SLH-3), to prevent them from being used by unpackers. In particular, packers can implement SLH by using the GOT/PLT hooking [27] or the inline hooking [5]. Since both of these approaches need to override the original code of each hooked method, packers will invalidate the instruction cache to ensure that the logic of each hooked method changes as expected.

Detection: Happer sets a HBRK to the handler that deals with the system call, *__NR_clear_cache*, and then retrieves the range of the virtual memory region, on which the cache flush operation is performed. If the region is included in the memory space of a system library, the SLH behavior is found.

(6) Dynamic Dex File Loading (DDL): Packers may load the hidden Dex file at runtime to prevent the packed app from being analyzed statically. Commonly, there are 3 ways to implement DDL. ① Packers initialize a class loader to perform the dynamic Dex file loading (DDL-1). ② Packers call the methods (e.g., *loadDex*) of the DexFile class to load the Dex file at runtime (DDL-2). ③ Packers call the methods (e.g., *makePathElements*) declared in the DexPathList class to dynamically load the hidden Dex file (DDL-3).

Detection: To identify DDL-1/2/3, Happer inspects the resolved method invocations to find whether the relevant system functions have been invoked to perform the Dex file loading.

(7) Dynamic Dex Data Modification (DDM): Many unpackers [7, 8, 86] adopt the one-pass unpacking mechanism [72] to retrieve the dynamically released Dex data. To evade being unpacked, packers may fill the original Dex items with invalid data, and dynamically recover them right before they are used by the Android runtime. There are 2 ways for packers to recover the valid Dex items at runtime. ① Since packers may load their native libraries before the release of the hidden Dex files, they can call *JNI_OnLoad* to recover the Dex items, e.g., header_item (DDM-1), class_def_items (DDM-2), and encoded_methods (DDM-3). ② Since *ClassLinker::LoadClass* is used by the ART runtime to parse the bytecode of the app methods defined in a specific app class, packers can leverage this method to recover the code_items (DDM-4).

Detection: To identify DDM-1/2/3, Happer sets a HBRK to the instruction address, where the execution of *JNI_OnLoad* just returns. To detect DDM-4, Happer sets a HBRK to *LoadClass*. Once a HBRK is hit, Happer compares the in-memory Dex file with that obtained when *DexFile::<init>* was called. If the Dex files' header_items, class_def_items, encoded_methods, or code_items are different, DDM behaviors are noticed.

(8) Dynamic Runtime Object Modification (DOM): In ART runtime, different types of Dex data are represented by various runtime objects. For example, a class_def_item is denoted by a mirror::Class object, and an encoded_method corresponds to an ArtMethod instance. Exploiting this observation, packers may adopt DOM to prevent the Dex data from being retrieved by unpackers. There are 2 ways for the packed app to modify the runtime objects. ① Packers hook *ClassLinker::LoadMethod* to modify the ArtMethod objects (DOM-1). ② Packers introduce or modify the static initialization method of each app class to

modify the relevant `mirror::Class` object (DOM-2).

Detection: `Happer` uses 2 approaches to identify DOM-1 and DOM-2, respectively. ① `Happer` sets a HBRK to the start address of *LoadMethod* and records its return address. Then, `Happer` sets another 2 HBRKs to the last instruction and the return address of *LoadMethod*, individually. When HBRKs are hit, values stored in fields `dex_code_item_offset_` and `access_flags_` of the related `ArtMethod` object are retrieved. If differences are found among the retrieved values, DOM-1 is detected. ② `Happer` sets a HBRK to the return address of *LoadMethod* and records the `ArtMethod` objects handled by this method. Then, `Happer` sets HBRKs to the start address of *ClassLinker::InitializeClass* and the address of a particular instruction in this method, at which the invocation to the static initialization method of an app class just returns. When HBRKs are hit, `Happer` retrieves the `ArtMethod` objects stored in the `methods_` field of the `mirror::Class` object, the second parameter of *InitializeClass*, and compares them with the previously recorded ones.

However, it is nontrivial to locate the `methods_` field due to the complicated data structure of `mirror::Class`. To tackle this issue, we leverage the observation that the `clinit_thread_id` field of the `mirror::Class` object stores the PID value of the packed app. Specifically, we locate the virtual memory address of `clinit_thread_id`, and then compute the address of the `methods_` field according to the offset between the 2 fields. If any `ArtMethod` object in `methods_` has been modified by the static initialization method of the app class, DOM-2 is found.

(9) Dex Data Fragmentation (DDF): Besides DDM and DOM, dispersing the Dex items of the loaded Dex file can evade the one-pass unpacking mechanism and prevent unpackers from dumping the valid Dex data. Specifically, since different types of Dex items are referenced according to their file offsets, packers can release the protected Dex file's `class_data_items` (DDF-1) or `code_items` (DDF-2) to separated memory regions.

Detection: To detect DDF behaviors, `Happer` accesses the in-memory Dex file when *DexFile::<init>* is invoked, and then examines the offset of each `class_data_item` and `code_item`. If an offset is out of the memory region that stores the Dex file, a DDF behavior is found.

(10) JNI Transformation (JNT): Bytecode stored in the Dex file can be reverse-engineered by disassemblers [13, 16, 21, 23, 34, 35]. Since it is more difficult to understand the semantic of native code compared with the bytecode, packers may use the native code to reimplement selected app methods, e.g., those inherited from the `Activity` class (JNT-1).

Detection: `Happer` examines the resolved method invocations to find whether the callbacks of app components' parent classes (e.g., *Activity.onCreate*) are invoked via JNI reflection. If so, the JNT-1 behavior is detected.

Remark: Identifying packing behaviors is important for unpacking. It empowers `Happer` to effectively (**R1**) and adaptively (**R3**) unpack the app. First, according to the identified packing behaviors, `Happer` can adopt various approaches to invalidate the anti-analysis techniques (in §VIII). Second, `Happer` can choose a proper way to efficiently unpack the app according to the recognized packing behaviors (in §VI-A). Third, according to the detected packing behaviors, `Happer` can employ the effective approaches to retrieve all the hidden Dex data in order to completely recover the Dex file (in §VI-B).

TABLE III: Dex data collection points.

Behavior	Type	Dex Data Collection Point
–	–	*addr*(`DexFile::<init>`, 0x0)
DDL	1-3	No additional Dex data collection point is required.
DDM	1-3	*addr*(`JavaVMExt::LoadNativeLibrary`, 0x75e)
	4	The return address of the `ClassLinker::LoadClass` method.
DOM	1	The return address of the `ClassLinker::LoadMethod` method.
	2	*addr*(`ClassLinker::InitializeClass`, 0x8d8)
DDF	1-2	No additional Dex data collection point is required.
	3	The same as DDM-2/3 and DOM-1/2.

VI. Unpacking Apps

Based on the identified packing behaviors, `Happer` chooses the proper collection points to retrieve the dynamically released Dex data (§VI-A and §VI-B), and then assembles the Dex data to a valid Dex file (§VI-C).

A. Determining Dex Data Collection Points

We define 2 types of Dex data collection points, one for obtaining the metadata (e.g., name, size, and start address) of the in-memory Dex file and another for collecting the Dex items of the Dex file. Table III lists the Dex data collection points for different packing behaviors, where the symbol *addr(method, offset)* refers to the instruction address located at the specified offset in a particular system function.

More specifically, since *DexFile::<init>* will be called by the ART runtime to create the `DexFile` object according to the header item of the in-memory Dex file, `Happer` collects the metadata when this method is invoked. Then, according to the start address and the size of the Dex file, `Happer` retrieves it from physical memory. Meanwhile, if the packed app has DDF-1/2 behaviors, `Happer` collects the dispersed Dex items at this point as well, because the in-memory Dex file contains the Dex items that hold the offsets for locating the dispersed Dex items. Moreover, if the packed app has DDM behaviors, `Happer` retrieves the recovered Dex data separately when *JNI_OnLoad* returns and the execution of *ClassLinker::LoadClass* finishes according to the implementations of DDM (in §V-C). In addition, if the packed app adopts DOM-1/2 to modify the ART runtime objects, `Happer` retrieves the related Dex data before the returns of *ClassLinker::LoadMethod* and *ClassLinker::InitializeClass*, because the modified ART runtime objects have already been filled with the valid data at these points.

B. Collecting Dex Data

To collect the hidden Dex data of the packed app, `Happer` needs to locate the memory regions storing these data. Meanwhile, since advanced packers may release the hidden Dex data right before the data is used by the Android runtime (**C3**), `Happer` forces the packer to release all the hidden Dex data for recovery. `Happer` locates the Dex data in virtual memory by determining the start address and the size of the target data, and

the start address is obtained or calculated according to particular register values or memory content at the Dex data collection points (e.g., the register `R1` points to the start address of the in-memory Dex file when *DexFile::<init>* is called), but the size of the Dex data cannot always be obtained directly. For instance, the size of the Dex file can be directly obtained according to the second parameter of *DexFile::<init>* and the size of each `class_def_item` is always 32 bytes, whereas the size of each `class_data_item` or `code_item` varies because their sizes are determined by concrete Dex items. Hence, `Happer` dynamically calculates the sizes of the Dex items to be dumped.

Specifically, since `class_data_item` is composed of the `uleb128` values (e.g., `encoded_field`) with distinct sizes, we calculate its size by summing the lengths of all the `uleb128` values. Since `code_item` consists of various fields (e.g., `insns`, `tries_size`, and `padding`) storing the data related to the bytecode of the ART method, we compute the size of `code_item` based on the lengths of all these fields. Particularly, `code_item` is 4-byte aligned and uses `padding` to meet the requirement. Consequently, if `insns` contains odd number of Dalvik bytecode and `tries_size` is non-zero, the size of `padding` is 2 bytes, otherwise, the size of `padding` is zero. Accordingly, `Happer` first gets the size of the `padding` field and then calculates the entire length of `code_item`.

Besides the task of locating the dynamically released Dex data, `Happer` will force the packed app to release all the hidden Dex data so that `Happer` can completely recover Dex file of the packed app. According to the packing behaviors disclosed in §V-C, packers with `DDM`-4 and `DOM`-1/2 behaviors release the hidden Dex data right before the data is used by the Android runtime. Unfortunately, to our best knowledge, none of existing unpackers uses an effective approach to ensure that they can retrieve all the hidden Dex data.

To force the packed app with `DDM`-4 and `DOM`-1/2 behaviors to release all the hidden Dex data, `Happer` makes the ART runtime load each method of the app and execute the static initialization method of each class of the app. `Happer` accomplishes this task through 3 steps. First, to collect the name of each app class, `Happer` parses each `class_def_item` of the Dex file retrieved when *DexFile::<init>* is invoked. Second, since *ClassLinker::FindClass* will call *ClassLinker::LoadMethod* to resolve the methods of an app class, `Happer` hooks *FindClass*. More specifically, to make the ART runtime load all the app methods, `Happer` feeds the name of each app class to *FindClass*. Third, since *ClassLinker::EnsureInitialized* will execute the static initialization method of an app class, `Happer` hooks this method. More specifically, `Happer` passes the name of each app class to *EnsureInitialized*, which makes the ART runtime execute the static initialization method of each app class.

C. Reassembling Dex Data

To recover the Dex file of the packed app, `Happer` assembles the collected Dex data through 2 steps.

First, `Happer` appends the collected Dex items to the end of the Dex file that is dumped when *DexFile::<init>* is called by the ART runtime. Meanwhile, `Happer` records the file offsets of the appended Dex items. Such offset information will be used by the next step to generate the valid Dex file.

Second, `Happer` leverages our customized `baksmali` [13] and `smali` [30] to reassemble the collected Dex data into a valid Dex file. More specifically, we feed the Dex file generated in the first step and the file offsets of the appended Dex items to `baksmali`. When `baksmali` transforms the Dex file's `class_def_items`, `class_data_items`, and `encoded_methods` to their internal representations (e.g., `DexBackedClassDef` and `DexBackedMethod`), the customized `baksmali` will refer to the input file offsets to locate the corresponding Dex items, because the appended Dex items contain the valid data for recovering the Dex file. After `baksmali` decompiles the Dex file, `Happer` uses `smali` [30] to conduct the recompilation and outputs the generated Dex file as the recovered Dex file of the packed app.

VII. Evaluation

We implement `Happer` with more than $7k$ lines of Java code and $10k$ lines of Python code based on `ds-5` [9]. `Happer` works as an external debugger of Juno r2 development board and supports both Android 6.0 and 8.1. We evaluate the performance of `Happer` by answering the following 5 research questions.

- **RQ1**: Can `Happer` identify more packing behaviours of popular commercial packers than other studies?
- **RQ2**: Can `Happer` recover the original Dex file of the app packed by popular commercial packers?
- **RQ3**: Can `Happer` facilitate the off-the-shelf static analysis?
- **RQ4**: What is the additional overhead incurred by `Happer`?
- **RQ5**: Is it easy to specify packing behaviors by using BDL?

Data Set: We use three data sets to evaluate `Happer`. The first set (*FSet*) contains 20 apps randomly selected from F-Droid [18], each of which are packed by 12 commercial packers. In detail, we uploaded the apps to 5 public packing service providers (i.e., Ali [4], Baidu [12], Ijiami [19], Qihoo [29], and Tencent [32]) in Nov. 2016 and then downloaded the packed apps. Note that Ali no longer provided the public packing service whereas 3 new vendors (i.e., Bangcle [14], Kiwi [24], and Testin [33]) are found to provide packing services in Nov. 2018. Therefore, we further uploaded the apps to these 7 packing services and then downloaded the packed samples. In total, there are 240 packed apps in *FSet*. It is worth noting that, although several packing service providers (e.g., Qihoo and Bangcle) support applying obfuscations to the input APK, we turn off these options to ease the cross-checking operation performed in §VII-B. The second set (*HSet*) contains 24,031 legitimate/commercial Android apps collected by a global smartphone manufacturer, and the last set (*MSet*) contains 1,787 malware samples provided by two leading security companies.

A. Identification of Packing Behaviors

We apply `Happer` to identifying packing behaviors of the apps in *FSet*, and the results are summarized in the left portion of Table IV. Meanwhile, in the right portion, we compare the behaviors identified by `Happer` with the behaviors reported

by 6 unpacking tools. Moreover, we use Happer to detect the packing behaviors of the apps in *HSet*.

(1) Commercial Packers: Here, we detail the behaviors of Baidu, Qihoo, and Tencent packers because they were available in both 2016 and 2018. More packing behaviors of Tencent, Ali, Bangcle, Ijiami, Kiwi, and Testin packers are presented in Appendix-G.

Baidu: Both Baidu-16 and Baidu-18 adopt ADG, DDL, and JNT to pack the app, while the former also employs DDM and DDF behaviors. ① To prevent the app from being analyzed by debuggers, both Baidu packers call *Debug.isDebuggerConnected* to detect the presence of JDWP-compliant debuggers [22] (ADG-1). ② Both packers dynamically load the hidden Dex file (DDL-1). Precisely, Baidu-16 calls *DexClassLoader.<init>* and Baidu-18 calls *InMemoryDexClassLoader.<init>* to achieve the purpose. ③ To protect the bytecode of critical methods, Baidu packers reimplement the inherited *onCreate* method in each activity component of the packed app (JNT-1). Besides these common behaviors, Baidu-16 also modifies the protected Dex file at runtime. ④ Baidu-16 modifies the header section of the released Dex file (DDM-1) after it has been resolved by the ART runtime. Precisely, `string_ids_off`, `method_ids_off`, and `class_defs_off` are filled with `0x0`. ⑤ Baidu-16 loads the `class_data_items` of the released Dex file to dispersed memory regions instead of contiguous memory space (DDF-1).

Qihoo: Both Qihoo packers have the similar behaviors. ① To protect the apps from being dynamically analyzed, Qihoo-16 and Qihoo-18 adopt TCK-2 to calculate the execution time of a specific native task, and the timeout threshold is set to one second. ② These packers call *DexFile.loadDex* to dynamically load the protected Dex files at runtime (DDL-2). ③ Both Qihoo packers reimplement the *onCreate* callback of each activity component of the packed app using native code (JNT-1).

(2) Comparison: To evaluate the performance of Happer in identifying packing behaviors, we compare Happer with 6 state-of-the-art studies, including DroidUnpack [45], TIRO [70], PackerGrind [72], AppSpear [53], and DexHunter [86], in terms of the observed packing behaviors. Since most of the unpackers under evaluation are either close-source or only partially open-source, we determine their capacities according to the experiment results reported in their research papers.

Results: The comparison results are shown in the right portion of Table IV. Note that the * suffix indicates that such the behavior has been noticed by the corresponding study but its details remain undisclosed. We can see that Happer totally identifies 27 packing behaviors in 10 categories, whereas other unpackers recognize only 7 packing behaviors and 6 categories at most. Specifically, compared with DroidUnpack, which aims at disclosing packing behaviors, Happer exposes 7 more categories and 19 more different packing behaviors. Moreover, among the identified 27 packing behaviors, the details about 17 packing behaviors (including ADG-1/3/4/5/6, AEU-1/2/3, TCK-2, SLH-1/3, DDL-1/3, DDM-2, DOM-2, and DDF-1/2) are disclosed for the first time. One possible reason is that previous studies mainly focus on unpacking apps rather than identifying packing behaviors. Additionally, the packers are evolving and their packing behaviors may also change.

(3) Packing Behaviors in Legitimate Android Apps: We apply Happer to analyzing the legitimate apps in *HSet* and find 1,710 packed samples, which can run on our platform and have at least one packing behavior. Then we count the number of apps that have the same packing behavior (Num_b) and the amount of samples that contain a common type of packing behavior (Num_t). Furthermore, we calculate the occurrence frequency of a specific type of packing behavior (i.e., $Ratio_l$ = $Num_t \div Num_b$) and the popularity of a particular packing behavior (i.e., $Ratio_g$ = $Num_b \div \#HSet$, where $\#HSet$ refers to the number of packed apps in *HSet*).

Results: The statistical results are shown in Table V. We notice that DDL is the most commonly adopted packing behavior. However, only a few packed apps adopt the sophisticated Dex data protection strategies, e.g., DDM, DOM, and DDF. Moreover, about a quarter of packed apps in *HSet* employ TCK, SLH, or JNT to defeat unpackers. Additionally, a portion of packed apps check their running environments to detect the presence of debuggers, emulators, or DBI frameworks.

(4) Packing Behaviors in Malicious Android Apps: We apply Happer to analyze the malware in *MSet*, of which 214 samples are monitored with at least one packing behavior.

Results: The statistical results are shown in Table VI. We discover that SLH, DDL, and ADG are the top three commonly adopted packing behaviors. However, only a few packed malware adopt the sophisticated Dex data protection approaches, e.g., DDF, DOM and DDM. Additionally, more than a quarter of packed apps in *MSet* employ TCK and JNT to defeat unpackers. Moreover, a portion of packed malicious apps will examine their running environments to detect the presence of debuggers, emulators, or DBI frameworks.

Answer to RQ1: Happer disclosed more packing behaviors than the previous studies. Among the 27 observed behaviors, the details about 17 of them are exposed for the first time.

B. Effectiveness of Unpacking

We apply Happer to unpacking the apps in *FSet*. Since we have the original apps of these packed samples, we compare each recovered Dex file with its corresponding original Dex file to assess whether Happer can unpack the app correctly. More specifically, we feed the recovered Dex file and the original Dex file to JEB [23], a commercial decompiler, and manually examine the decompilation results to identify the difference.

Results: Happer can completely recover the original Dex files of the apps packed by 7 of the 12 commercial packers under evaluation, including Ali-16, Bangcle-18, Ijiami-16/18, Tencent-16/18, and Kiwi-18. However, existing unpackers can at most completely recover the apps packed by 4 of the commercial packers, including Ali-16, Bangcle-18, Tencent-16/18. For the apps packed by Ijiami-16/18 or Kiwi-18, existing unpackers fail to completely recover their Dex files because the existing unpackers cannot retrieve all the hidden Dex data released

TABLE IV: An overview of behaviors of commercial Android packers. Refer to Table II for the description of these behaviors.

Packing Behaviour	2016					2018							Other Unpackers[1]					
	Baidu	Ijiami	Qihoo	Tencent	Ali	Baidu	Ijiami	Qihoo	Tencent	Bangcle	Kiwi	Testin	DU-18	TR-18	AS-18	PG-17	AS-15	DH-15
ADG	ADG-1	ADG-1/3-6	–	ADG-2	–	ADG-1	ADG-1/3-6	–	–	ADG-2	–	–	–	–	ADG-2	ADG-2	–	ADG-2
AEU	–	AEU-1/2/3	–	–	–	–	AEU-1/2/3	–	–	–	–	–	–	–	–	–	–	–
ADI	–	ADI-1	–	–	–	–	ADI-1	–	–	–	–	–	–	–	–	ADI-1	–	–
TCK	–	TCK-1	TCK-2	–	–	–	TCK-1	TCK-2	–	–	–	TCK-1	–	–	TCK-1	TCK-1	–	–
SLH	–	SLH-1/3	–	–	–	–	SLH-1/3	–	–	SLH-1/2	–	–	SLH-2	–	–	–	–	–
DDL	DDL-1	–	DDL-2	DDL-2	DDL-2	DDL-1	–	DDL-2	DDL-2	DDL-3	–	DDL-3	DDL-*	DDL-2	DDL-*	DDL-*	DDL-*	DDL-*
DDM	DDM-1	DDM-4	–	DDM-1	DDM-2/3	–	–	–	–	–	–	–	DDM-*	DDM-3	DDM-*	DDM-1/3	DDM-1/4	DDM-*
DOM	–	–	–	–	–	–	DOM-1	–	–	–	DOM-2	–	–	DOM-1	–	–	–	–
DDF	DDF-1	–	–	–	–	–	DDF-2	–	–	–	DDF-2	–	–	–	DDF-*	–	DDF-*	–
JNT	JNT-1	–	JNT-1	–	–	JNT-1	–	JNT-1	–	–	–	JNT-1	–	JNT-1	JNT-1	JNT-1	–	–
#Total Behaviors	27 different packing behaviors are monitored by Happer.												3	4	6	7	4	3

[1] DU-18 (DroidUnpack [45]), TR-18 (TIRO [70]), AS-18 (AppSpear [53]), PG-17 (PackerGrind [72]), AS-15 (AppSpear [79]), DH-15 (DexHunter [86]).

TABLE V: Packing behaviors found in legitimate packed apps.

Behavior	Type	Ratio$_l$	Ratio$_g$	Behavior	Type	Ratio$_l$	Ratio$_g$
ADG	1	104/289 (36.0%)	289/1710 (16.9%)	SLH	1	280/389 (72.0%)	389/1710 (22.7%)
	2	185/289 (64.0%)			2	300/389 (77.1%)	
	3	91/289 (31.5%)			3	280/389 (72.0%)	
	4	90/289 (31.1%)		DDL	1	99/1614 (6.1%)	1614/1710 (94.4%)
	5	90/289 (31.1%)			2	1283/1614 (79.5%)	
	6	90/289 (31.1%)			3	234/1614 (14.5%)	
AEU	1	81/81 (100.0%)	81/1710 (4.7%)	DDF	1	0/35 (0.0%)	35/1710 (2.0%)
	2	81/81 (100.0%)			2	35/35 (100.0%)	
	3	81/81 (100.0%)			N/A	N/A	
ADI	1	87/87 (100.0%)	87/1710 (5.1%)	JNT	1	440/440 (100.0%)	440/1710 (25.7%)
TCK	1	83/485 (17.1%)	485/1710 (28.4%)	DDM	1	2/47(4.2%)	47/1710 (2.7%)
	2	402/485 (82.9%)			2	0/47(0.0%)	
DOM	1	35/35 (100.0%)	35/1710 (2.0%)		3	0/47(0.0%)	
	2	0/35 (0.0%)			4	45/47(95.8%)	

TABLE VI: Packing behaviors found in malicious packed apps.

Behavior	Type	Ratio$_l$	Ratio$_g$	Behavior	Type	Ratio$_l$	Ratio$_g$
ADG	1	64/109 (58.7%)	109/214 (50.9%)	SLH	1	135/165 (81.8%)	165/214 (77.1%)
	2	45/109 (41.3%)			2	45/165 (27.3%)	
	3	49/109 (45.0%)			3	129/165 (78.2%)	
	4	48/109 (44.0%)		DDL	1	80/157 (51.0%)	157/214 (73.4%)
	5	48/109 (44.0%)			2	32/157 (20.4%)	
	6	48/109 (44.0%)			3	45/157 (28.7%)	
AEU	1	49/49 (100.0%)	49/214 (22.9%)	DDF	1	11/52 (21.2%)	52/214 (24.3%)
	2	49/49 (100.0%)			2	41/52 (78.8%)	
	3	49/49 (100.0%)			N/A	N/A	
ADI	1	49/49 (100.0%)	49/214 (22.9%)	JNT	1	83/83 (100.0%)	83/214 (38.8%)
TCK	1	49/61 (80.3%)	61/214 (28.5%)	DDM	1	0/13(0.0%)	13/214 (6.1%)
	2	12/61 (19.7%)			2	5/13(38.5%)	
DOM	1	41/41 (100.0%)	41/214 (19.2%)		3	5/13(38.5%)	
	2	0/41 (0.0%)			4	8/13(61.5%)	

at runtime. Additionally, since Baidu-16/18, Qihoo-16/18, and Testin-18 reimplement specific app methods using native code, the Happer-recovered Dex files of the apps packed by these 5 packers lack the bytecode of those reimplemented app methods because their bytecode will never be released to the memory. Note that none of existing unpackers can tackle this issue.

We also run Happer and DroidUnpack [45] to unpack the 1,924 packed apps from *HSet* and *MSet*, including 1,710 legitimate and 214 packed malware samples, respectively. From the results, we find that Happer successfully unpacks all these samples and outputs valid Dex files containing the hidden Dex data. To verify the validations of the produced Dex files, we diassemble them using the off-the-shelf Dex file disassembling

TABLE VII: The Analysis on Malware Samples.

Packers	Ali	Baidu	Bangcle	Ijiami	Kiwi	Qihoo
$\#App$	5	80	45	49	7	28
PS_{avg}	0.0	0.0	0.5	0.7	5.9	0.6
RS_{avg}	11.6	10.9	4.3	7.4	8.6	14.6
ΔS_{avg}	+11.6	+10.9	+3.8	+6.7	+2.7	+14.0

tools, including Baksmali, DexDump, Jadx, and IDA Pro, which adopt different strategies to check the Dex file, and all these Dex files are disassembled normally. Meanwhile, DroidUnpack dumps Dex files from 822 and 92 samples of *HSet* and *MSet* respectively. After verification, 17 packed legitimate apps and 16 packed malware are successfully unpacked. However, the dumped Dex files for other samples are the shell Dex files inserted by the packers or other class files (i.e., webview). By studying its source code, we find that the failures are mainly caused by the following reasons. First, it mainly focuses on studying the packing techniques from 2010 to 2015, thereby cannot handle the recent packers. Second, it looks up the Dex data from their corresponding Android runtime representations (e.g., the ArtMethod instances), but the advanced packers (e.g., Ijiami and Kiwi) dynamically modify them to hinder unpacking. Third, it does a lot of time-consuming work at runtime (e.g., memory instrumentation and query operations), which can result in app timeouts and crashes.

Answer to RQ2: Happer outperforms existing unpackers in terms of unpacking the apps because it can retrieve all the hidden Dex data released to the memory.

C. Assistance to Static Analysis

We first use Happer to unpack the 214 packed malware in *MSet* and output the recovered Dex files. Since the majority of static analysis tools [39, 40, 57, 66, 71, 80–83, 85] detect malware by finding invocations of sensitive framework APIs, we use FlowDroid [39] to find sensitive APIs presented in the recovered Dex files and the corresponding packed ones.

Results: Table VII shows the analysis results, where $\#App$ indicates the number of packed malware. PS_{avg} and RS_{avg} denote the average number of sensitive APIs found in the packed Dex files and the recovered Dex files, respectively. ΔS_{avg} represents the increased number of sensitive APIs found in the recovered Dex files (i.e., $\Delta S_{avg} = RS_{avg} - PS_{avg}$).

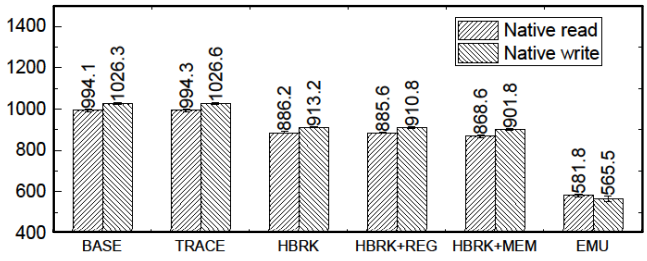


Fig. 6: The performance of Happer.

The results show that more sensitive APIs are exposed in the recovered Dex files. For example, there are almost no sensitive APIs in malware packed by Ali, Baidu, Bangcle, Ijiami, and Qihoo, whereas many sensitive APIs are found in the corresponding recovered Dex files.

Answer to RQ3: Happer can assist the off-the-shelf static analysis to expose more stealthy behaviors of the packed apps by providing the recovered Dex files.

D. Overhead

The extra overhead introduced by Happer may come from 4 aspects, including the instruction tracing, the HBRK suspension, the register accessing, and the memory data dumping. To evaluate the slowdown incurred by these operations, we instruct Happer to perform only an individual operation in one test. For each operation, we run CF-Benchmark [15] 30 times and calculate the average performance score. The score computed when we disable all the functions of Happer is treated as the baseline (i.e., the BASE bar in Fig.6). Note that a higher score denotes a better performance.

Results: When we enable the tracing function, the performance score is indicated by the TRACE bar and almost no slowdown is incurred. The slowdown incurred by HBRKs is indicated by the HBRK bar, and around 10% additional slowdown is incurred. Moreover, to asses the slowdown incurred by accessing the register and dumping the memory content, we instruct Happer to get the values of registers and retrieve the app library libCFBench.so that has been loaded to the memory by CF-Benchmark. The scores of these operations are denoted by the bars, HBRK+REG and HBRK+MEM, respectively.

In addition, we compare the performance of Happer with Qemu, which is used by the emulator-based dynamic analysis, and its performance score is denoted by the EMU bar. During measurement, Qemu runs Android 6.0 as the guest system, and the host is equipped with Intel i7-6700k CPU and 32GB RAM. According to the results, the emulator achieves only about half of the BASE score, indicating that the emulator-based approaches will incur more slowdown than Happer.

Answer to RQ4: By leveraging ARM's hardware features, Happer incurs reasonable slowdown to the tracked app.

E. Efforts for Composing Behavior Specification

To evaluate the efforts of writing the behavior specification by using BDL, we invite five fresh graduate students majored in computer science to write the specifications for four commonly adopted packing behaviors in four categories, including ADG-2, AEU-1, SLH-2, and DDL-2.

In detail, to let the participants know how to write the specification using BDL, we provide them an introduction to the grammar of BDL and an example specification for SLH-1. On average, they took around 3.5 minutes to preliminary understand BDL. In addition, we explain the technical details about the selected packing behaviors to the participants and calculate the time spent by the participants to write each specification. On average, they can accurately write the behavior specification for ADG-2, AEU-1, SLH-2, and DDL-2 in around 3.9, 5.6, 1.8, and 3.4 minutes, respectively. More time is needed for writing the behavior specification for AEU-1 because it is the most complicated one among the four selected packing behaviors. Note that, each of these specifications consists of no more than 10 BDL statements. Moreover, we provide the participants the necessary source code of Happer and ask them to extend the tool in order to make it support the identification of AEU-1 by directly customizing the source code. Within an hour, only two participants have submitted their attempts but none of them can achieve the purpose.

Answer to RQ5: BDL can assist analysts in their unpacking tasks by significantly simplifying the extension of Happer to support the monitoring of new packing behaviors.

VIII. Discussion

As we just use Happer to analyze the benign packed apps for research purpose, we discuss its limitations from the following three aspects.

Transparency: We introduce the potential fingerprints that might be exploited to detect Happer and our countermeasures. The first potential fingerprint is the ETM registers used by Happer, which can be monitored in kernel layer (**F1**). Since the kernel module loadable option is disabled when building the guest system and we assume that the apps are unable to gain the root privilege, the packers cannot detect Happer by using **F1**. The second type of potential fingerprint is the slowdown incurred by the HBRK suspension during dynamic instrumentation (**F2**). To prevent packed apps from detecting Happer through **F2**, we design a specific mechanism to detect the slowdown checking behavior performed by the packed app and bypass it by returning modified timestamps to the app when such a behavior is found. Thus, it is hard for packed apps to discover Happer by using **F2**. The third type of potential fingerprint is the special system properties that are different from those in the official Android images (**F3**). To remove such fingerprints, we let the system have the same properties as those in the official system for commercial smartphones (e.g., Pixel) when building the guest system for Juno. Hence, packed apps cannot know that they are running on a hardware platform for analysis by checking system properties (i.e., **F3**).

Detection of Novel Packers: `Happer` supports all the packing behaviors of the evaluated apps, which are much more than the behaviors reported by previous studies and unpackers. When encountering new packers, `Happer` can get the memory regions used by the app and then retrieve the virtual addresses of the executed code to detect the instinct packing behavior "write-then-execute", which is also exploited by DroidUnpack [45], to identify the new packing behaviors according to the its dynamically tracked information. However, for other packing behaviors, manual effort may be required to determine them. The dynamically tracked information, including instructions, memory data, method invocations, and dumped Dex files, can assist in such a determination procedure. Then, `Happer` can be extended to support the new packing behaviors using BDL timely. Note that such manual effort is just required once for all apps packed by the same packer. In the future, we will further improve `Happer` by designing reinforcement learning based algorithms to automatically learn the new packing behaviors.

Scalability: Since the ARM platform only has six HBRKs [11], at most six packing behaviors, whose detection requires HBRK, can be identified by `Happer` at a time. Hence, to identify all an app's packing behaviors, `Happer` needs to run the app multiple times. In future work, we will design an algorithm to efficiently allocate HBRK in order to increase the number of packing behaviors that can be identified by `Happer` at a time.

IX. Related Work

In this section, we present the most related work on native (un)packers, Android unpackers, and dynamic app analysis.

PC (Un)packers: There are already a number of studies related to packing or unpacking but most of them focus on native/PC programs [41, 48, 49, 60, 63, 68], and none of them exploited hardware features like PT for x86 or ETM for ARM to unpack programs. For example, PolyUnpack [62] monitors the program inside a debugger and then determines the packed code by comparing the dynamically traced instruction sequence with the static disassembly of the program. Omniunpack [54] tracks the memory modifications at page level to determine the hidden code of packed binaries. Renovo [49] and the unpacker presented in [68] leverage the whole-system emulation techniques (i.e., TEMU) to unpack the packed binaries. Saffron [60] conducts dynamic unpacking tasks leveraging Intel's PIN.

However, these unpacking techniques and tools cannot be applied to unpacking Android apps directly due to the different execution mechanisms of the unpacking targets, namely native instructions and Dex files. More precisely, the PC unpackers only concerned with the native instructions executed by CPU, but Android unpackers need to consider the Dex items and Dalvik bytecode executed by Android runtime [45, 70].

Android Unpackers: With the growing adoption of packing techniques to evade analysis and detection, several Android app unpackers have been proposed [45, 56, 70, 72, 76, 79, 86]. However, most of them unpack the app according to the developer's experience on packers [79, 86], and thus they are ineffective in unpacking evolved packers. Meanwhile, although some unpackers will analyze packing behaviors [45, 70, 72], they rely on DBI, VMI, or the system modification, which can be detected and impeded by packers with anti-analysis abilities [59, 61, 66]. Moreover, existing unpackers usually incur much overhead because they need to emulate the entire Android system, instrument or track each executed instruction (or bytecode) [45, 56, 72, 76].

Hardware-assisted Dynamic Trace: Various hardware-assisted approaches [42, 43, 65, 84] have been proposed for program debugging and tracing on x86 platform by leveraging the hardware features, including hardware virtualization, hardware sensors, and PMU (Performance Monitoring Unit). Recently, a few hardware-assisted analysis methods have been proposed for ARM platform. For instance, Ninja [55] is a transparent malware analysis framework, and it implements the tracing and debugging functionalities via the hardware-assisted isolation execution environment TrustZone and the hardware features PMU and ETM. In addition, HART [44] is a kernel module tracing framework implemented based on ETM. However, both the implementation mechanisms and purposes of `Happer` are different from them.

Specifically, Ninja focuses on tracing and debugging ARM processes and HART is a kernel-specific tracing framework, whereas `Happer` focuses on analyzing and unpacking Android apps. Additionally, Ninja and HART work as the on-device tracing and debugging tools, but `Happer` consists of an online tracing component for collecting runtime information and an offline component for determining behaviors and assembling unpacking results. Moreover, `Happer` has a novel resolver for parsing the ETM stream, and it supports customized extensions using the provided behavior description language (i.e., BDL). Consequently, `Happer` is a novel hardware-assisted unpacking and analyzing approach proposed in this paper.

X. Conclusion

We propose a novel hardware-assisted unpacking tool `Happer` to deal with the ever-evolving Android packers. It leverages hardware features to monitor the packing behaviors and then selects the proper strategies to unpack the packed apps. The comprehensive evaluation shows that `Happer` observed much more packing behaviors than existing studies and effectively recovered the Dex files of the packed apps.

Acknowledgment

We sincerely thank Prof. Heng Yin for shepherding our paper and the anonymous reviewers for their constructive comments. We thank Luyi Yan for his assistance during paper revision. This work is partly supported by Hong Kong RGC Projects (No. 152223/17E, 152239/18E, CityU C1008-16G), NSFC Young Scientists Fund (No. 62002306), NSFC General Fund (No. 61872438, 61772371, 62002151, 61972332), HKPolyU Start-up Fund (ZVU7), Research Grant from Huawei Technologies, CCF-Tencent Open Research Fund, the Fundamental Research Funds for the Central Universities (No. K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), and Zhejiang Key R&D (No. 2019C03133).

REFERENCES

[1] "Android emulator detection," https://blog.trustlook.com/bangle-android-app-packer-unpacking/, 2018.

[2] "Bangle android app packer: Unpacking & analysis," https://blog.trustlook.com/bangle-android-app-packer-unpacking/, 2018.

[3] "drizzleDumper," https://github.com/DrizzleRisk/drizzleDumper, 2018.

[4] "Ali packer," https://security.alibaba.com, 2019.

[5] "Android arm inline hook," https://github.com/ele7enxxh/Android-Inline-Hook, 2019.

[6] "Android malware is becoming more resilient," https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient, 2019.

[7] "android-unpacker," https://github.com/strazzere/android-unpacker, 2019.

[8] "android_unpacker," https://github.com/CheckPointSW/android_unpacker, 2019.

[9] "Arm ds-5," https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio, 2019.

[10] "Arm ptm decoder, and arm etmv4 decoder," https://github.com/hwangcc23/ptm2human, 2019.

[11] "Armv8-a architecture reference manual," https://static.docs.arm.com/ddi0487/db/DDI0487D_b_armv8_arm.pdf, 2019.

[12] "Baidu packer," http://app.baidu.com/, 2019.

[13] "baksmali," https://github.com/JesusFreke/smali, 2019.

[14] "Bangcle packer," https://www.bangcle.com/, 2019.

[15] "Cf-bench," http://bench.chainfire.eu/, 2019.

[16] "Dex2jar," https://github.com/pxb1988/dex2jar, 2019.

[17] "Embedded trace macrocell specification," https://static.docs.arm.com/ihi0064/d/IHI0064D_etm_v4_architecture_spec.pdf, 2019.

[18] "F-droid," https://f-droid.org/, 2019.

[19] "ijiami packer," http://www.ijiami.cn/, 2019.

[20] "Implementing art just-in-time (jit) compiler," https://source.android.com/devices/tech/dalvik/jit-compiler, 2019.

[21] "Jadx," https://github.com/skylot/jadx, 2019.

[22] "Jdwp debugger," https://stuff.mit.edu/afs/sipb/project/android/docs/tools/debugging/index.html, 2019.

[23] "Jeb decompiler for android," https://www.pnfsoftware.com/jeb/android, 2019.

[24] "kiwisec packer," https://cloud.kiwisec.com, 2019.

[25] "oatdump," https://android.googlesource.com/platform/art/+/kitkat-dev/oatdump/oatdump.cc, 2019.

[26] "objdump," https://developer.android.com/ndk, 2019.

[27] "A plt hook library for android native elf," https://github.com/iqiyi/xHook, 2019.

[28] "Qemu, an open source processor emulator," https://www.qemu.org/, 2019.

[29] "Qihoo packer," http://dev.360.cn/, 2019.

[30] "smali," https://github.com/JesusFreke/smali, 2019.

[31] "System and kernel security for android," https://source.android.com/security/overview/kernel-security, 2019.

[32] "Tencent packer," https://intl.cloud.tencent.com/, 2019.

[33] "Testin packer," https://www.testin.cn/, 2019.

[34] "A tool for reverse engineering android apk files," https://ibotpeaches.github.io/Apktool/, 2019.

[35] "A tool for translating dalvik bytecode to equivalent java bytecode," https://github.com/Storyyeller/enjarify, 2019.

[36] "Zjdroid," https://github.com/halfkiss/ZjDroid, 2019.

[37] "ANTLR," https://github.com/antlr/antlr4, 2020.

[38] "Principles of ARM Memory," http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf, 2020.

[39] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[40] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. ACM/IEEE ICSE*, 2015.

[41] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proc. ACM CCS*, 2015.

[42] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *ACSAC*, 2013.

[43] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. ACM CCS*, 2008.

[44] Y. Du, Z. Ning, J. Xu, Z. Wang, Y.-H. Lin, F. Zhang, X. Xing, and B. Mao, "Hart: Hardware-assisted kernel module tracing on arm," in *Proc. ESORICS*, 2020.

[45] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un)packers: a systematic study based on whole-system emulation," in *Proc. NDSS*, 2018.

[46] C. Emanuele, G. Mariano, F. Yanick, and B. Davide, "Understanding linux malware," in *Proc. IEEE S&P*, 2018.

[47] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection." in *Proc. NDSS*, 2003.

[48] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proc. RAID*, 2008.

[49] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. ACM WORM*, 2007.

[50] I. Kirillov, D. Beck, P. Chase, and R. Martin, "Malware attribute enumeration and characterization," *The MITRE Corporation [online, accessed Apr. 8, 2019]*, 2011.

[51] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Proc. IEEE SCAM*, 2009.

[52] A. Kurmus and R. Zippel, "A tale of two kernels: Towards ending kernel hardening wars with split kernel," in *Proc. CCS*, 2014.

[53] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "AppSpear: Automating the hidden-code extraction and reassembling of packed android malware," *Journal of Systems and Software (JSS)*, vol. 140, pp. 3–16, 2018.
[54] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Proc. ACSAC*, 2007.
[55] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proc. USENIX Security*, 2017.
[56] ——, "DexLego: Reassembleable Bytecode Extraction for Aiding Static Analysis," in *Proc. DSN*, 2018.
[57] C. Qian, X. Luo, L. Yu, and G. Gu, "Vulhunter: Towards discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, 2015.
[58] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proc. DSN*, 2014.
[59] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su, "A framework for understanding dynamic anti-analysis defenses," in *Proc. ACM PPREW*, 2014.
[60] D. Quist and V. Smith, "Covert debugging: Circumventing software armoring," in *Proc. BlackHat*, 2007.
[61] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.
[62] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the hidden-code extraction of unpack-executing malware," in *Proc. ACSAC*, 2006.
[63] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, 2016.
[64] Y. Shao, X. Luo, and C. Qian, "Rootguard: Protecting rooted android phones," *Computer*, vol. 47, no. 6, 2014.
[65] C. Spensky, H. Hu, and K. Leach, "Lo-phi: Low-observable physical host instrumentation for malware analysis." in *Proc. NDSS*, 2016.
[66] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, 2017.
[67] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *Proc. NDSS*, 2015.
[68] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE S&P*, 2015.
[69] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proc. ASIACCS*, 2014.
[70] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in Android with TIRO," in *Proc. USENIX Security*, 2018.
[71] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim, "Toward engineering a secure android ecosystem: A survey of existing techniques," *ACM Computing Surveys*, 2016.
[72] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. ICSE*, 2017.
[73] L. Xue, X. Ma, X. Luo, L. Yu, S. Wang, and T. Chen, "Is what you measure what you expect? factors affecting smartphone-based mobile network measurement," in *Proc. IEEE INFOCOM*, 2017.
[74] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *Proc. IWQoS*, 2015.
[75] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, 2019.
[76] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Trans. on Software Engineering*, 2020.
[77] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. USENIX Security*, 2017.
[78] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *Proc. USENIX Security*, 2012.
[79] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware," in *Proc. RAID*, 2015.
[80] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. Leung, "Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies," *IEEE Trans. on Software Engineering*, 2019.
[81] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proc. DSN*, 2016.
[82] L. Yu, X. Luo, C. Qian, S. Wang, and H. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Trans. on Software Engineering*, 2018.
[83] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Towards automatically generating privacy policy for android apps," *IEEE Trans. on Information Forensics and Security*, 2017.
[84] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proc. IEEE S&P*, 2015.
[85] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014.
[86] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed Android applications," in *Proc. ESORICS*, 2015.

APPENDIX

A. The Used ETM and MMU Registers

Table VIII summarizes the ETM registers used for enabling

and configuring the tracing functionality, and Table IX lists the MMU registers used for fetching the memory data.

B. Calculating Physical Memory Address

There are a series of address translation (AT) instructions for translating a virtual address to its corresponding physical address in the format of "AT <at_op>, <Xt>", where at_op and Xt represent the translation operation and the virtual address, respectively. For instance, "AT S12E0R <Xt>" performs the stage 1 and stage 2 address translation (i.e., the virtual address is first converted to an intermediate physical address and then translated to the physical address) for the virtual address in EL0, and the corresponding physical address is stored in the physical address register, PAR_EL1 [11].

Accordingly, Happer leverages AT instructions to obtain the physical address of the demanded in-memory data according to its virtual address. Before getting the physical address from PAR_EL1, Happer checks whether the address translation has been successfully conducted according to the F field of PAR_EL1. If so (i.e., the value of the F field equals 0x0), Happer retrieves the calculated physical address from the PA field of PAR_EL1. Note that the physical address PA' read from the PAR_EL1 register is page-aligned, and thus Happer further converts it to the actual physical address PA of the target data through Equation 1, where VA and PS refer to the virtual address of the memory data and the size of the memory page, respectively.

$$PA = PA' + (VA \bmod PS) \tag{1}$$

TABLE VIII: The leveraged ARM ETM trace registers.

ETM Registers	Field	Purpose
TRCCONFIGR (Trace Configuration Register)	CID	Enable context ID tracing
	BB	Enable broadcast tracking
TRCCIDCCTLR0 (Context ID Comparator Control Register 0)	COMP0	Perform Context ID comparison with relevant byte in TRCCIDCVR0
TRCCIDCVR0 (Context ID Comparator Value Register 0)	VALUE	Specify the PID to be compared with during Context ID tracing
TRCIDR0 (ID Register 0)	TRCBB	Check if branch broadcast tracing is supported
TRCBBCTLR (Branch Broadcast Control Register)	RANGE	Specify address range comparator pair for branch broadcast tracing
TRCACVR0/TRCACVR1 (Address Comparator Value Registers 0/1)	ADDRESS	Specify the virtual memory range for branch broadcast tracing

TABLE IX: The leveraged ARM MMU registers/descriptor.

Register/Descriptor	Field	Purpose
PAR_EL1 (Physical Address Register (EL1))	F	Check if address translation succeeds
	PA	Obtain the base physical address PA_{base}
TTBR0_EL1 (Translation table base register(EL1))	BADDR	Obtain the translation table base address
Page descriptor	UXN	Check unprivileged execute permission
	AP	Check access permission

C. Inserted Memory Loading Instructions

We show the instructions for loading the data in Fig.7, where *memory_force_loading* takes in the virtual address and the size of the demanded data. Specifically, we use the LDR instruction

```
1 procedure memory_force_loading(addr, size) {
2   ; The register r0 stores the virtual memory address of the file to be loaded.
3   ; The register r1 stores the size of the entire file.
4   push  {r0, r1, r2, r3}          ; Store the registers r0-r3
5   mov   r2, r1 / PAGE_SIZE + 0x1  ; Calculate time for loading the whole file
6   mov   r3, #0x0                  ; Store the loop index into register r3
7   ldr   r1, [r0]      <           ; Force loading the data addressed by r0
8   add   r0, PAGE_SIZE |           ; Adjust to the next memory access point
9   add   r3, 0x1       |           ; Increase the loop count.
10  cmp   r3, r2        |           ; Judge whether the task has been finished
11  bne   {pc} - #0xc   >           ; To load next memory page
12  pop   {r0, r1, r2, r3}          ; Restore the registers r0-r3.
13 }
```

Fig. 7: The instrumentation code for loading the data.

to access each virtual memory slice of the data. By doing so, the OS completely loads the target data to physical memory.

D. Constructing Address-Function Map

Happer retrieves the memory-map information of loaded system files (i.e., system libraries and framework Oat files) from the */proc/*#pid*/maps* file. In this process, we find the executable memory region (i.e., VA_{exec}) for each loaded system library and framework Oat file. Then, we use objdump [26] and oatdump [25] to disassemble the system libraries and the framework Oat files. Meanwhile, we record the file offset (i.e., FO_{inst}) of each decompiled instruction.

However, the virtual address (i.e., VA_{inst}) of an instruction does not always equal to the result of adding FO_{inst} to VA_{exec} because the file offset (i.e., FO_{inst}) and the virtual offset (i.e., VO_{inst}) for an instruction are commonly inconsistent. To address this issue, we calculate the difference (i.e., δ_{gap}) between the 2 types of offsets through Equation 2. Precisely, for instructions in the system library, Happer uses objdump to retrieve the virtual memory offset (i.e., VO_{text}) and the file offset (i.e., FO_{text}) of the library's .text section. Meanwhile, for instructions in the framework Oat file, Happer uses oatdump to obtain the file offset of the Oat file's .oatexec section (i.e., FO_{exec}). Afterwards, VA_{inst} is calculated using Equation 3. Subsequently, we store the calculated virtual instruction address (i.e., VA_{inst}) and its corresponding system method (i.e., $method$) into the structure $map\{VA_{inst} \mapsto method\}$, which will be used to facilitate the identification of the system methods invoked by the packed app.

$$\delta_{gap} = \begin{cases} FO_{text} - VO_{text} & (System\ \ libraries) \\ 0 - FO_{exec} & (Oat\ \ files) \end{cases} \tag{2}$$

$$VA_{inst} = VA_{exec} + (FO_{inst} + \delta_{gap}) \tag{3}$$

E. Identifying Special Function Invocation

Since packers may hook system library methods to prevent the memory data from being dumped by unpackers, to disclose this packing behavior, Happer identifies the hooked methods invoked by the packed app and the details are shown in Fig. 8, in which *identify_special_invocation* takes each instruction address (i.e., $address$) recorded in the ETM stream and the address-method mapping (i.e., map) as the inputs.

Specifically, to recover the call stack for each executed instruction, we use a stack (i.e., *stack*) to record each system method invoked by the app. In detail, in Line 4, every invoked

```
   input : address is the address of an instruction recorded by the ETM;
           map is the mapping from instruction to method.
   output: hook, the set containing the hooked methods invoked by the targ
1  Function identify_special_invocation(address, map):
2      if address is the start address of a system method then
3          // An invocation to the system method.
4          method = map[address]; stack.push(method)
5      else if (method = map[address]) != NULL then
6          // An invocation returns to the system method.
7          last_method = stack.pop(); popup_method = last_method
8          while popup_method != method do
9              if popup_method == APP* and last_method == APP* then
10                 // An invocation to the hooked method is found.
11                 type = stack.peek(0) == method ? CALLER : CALLEE
12                 hook.add(<stack.peek(0), type>)
13             end
14             popup_method = stack.pop()
15         end
16         stack.push(method)
17     else
18         // An invocation to the application method.
19         if stack.peek(0) != APP* then
20             stack.push(APP)
21         end
22     end
23 return
```

Fig. 8: Identifying invocations to hooked methods.

system method is pushed into *stack*. Similarly, in Line 20, each invoked app method is pushed into *stack*. Moreover, since it is hard to decompile the packed app in advance due to the code protections adopted by the packer, we introduce the APP method to represent the app-defined method.

After recovering the runtime call stack, Happer determines whether a system library method is hooked based on the heuristic: the customized code of the packed app is first executed and then the hooked method is invoked by the customized code. In detail, in Line 6-16, Happer examines *stack* to check whether a hooked method has been invoked by the packed app. If the invocation returns from the app-defined method to the system method and there are more than one APP method presented in *stack*, we think that a hooked method has been invoked and the target of the returned invocation is the hooked method. However, we notice that the target identified through this way may be the caller of the hooked method. Accordingly, to locate the exact one, we consecutively pop up the method in the call stack until an APP method is found. Then, the caller of this APP method is considered as the hooked method, which will be added into the *hook* set.

F. Behavior Description Language (BDL)

BDL: BDL statements are constituted by BDL expressions. As shown in Fig. 5, four types of expressions are defined, including *basicExpr*, *condExpr*, *funcExpr*, and *exprList*, which specifies a basic expression (e.g., a variable), a condition checking, a BDL function invocation, and an expression list, respectively. Note that *exprList* is used to specify the parameters of a BDL function. There are also four lexical items defined. In particular, *NEWLINE*, *INTCONSTANT*, *STRINGCONSTANT*, and *IDENTIFIER* refers to the end of a line, the integer constant, the literal constant, and the variable, respectively. Note that, to simplify the grammar specification, BDL conditions (*condExpr*) includes relational expressions that connect two expressions via ==, !=, >, and <, and logical expressions that connect two expressions via && and ||.

```
01 // get the first parameter value of the fopen function in libc.so
02 var fopenArg1 <- getArgValue("libc::fopen", 1);
03 // get the first parameter value of the strncmp function in libc.so
04 var strncmpArg1 <- getArgValue("libc::strncmp", 1);
05 // get the second parameter value of the strncmp function in libc.so
06 var strncmpArg2 <- getArgValue("libc::strncmp", 2);

07 // if the following two conditions are satisfied, AEU-1 is found
08 var detected <- 0;
09 if (fopenArg1 == "/proc/tty/drivers"
    && (strncmpArg1 == "goldfish" || strncmpArg2 == "goldfish")) {
10     detected <- 1;
11 }
```

Fig. 9: The behavior specification for AEU-1.

The core for BDL is the defined BDL function (*funcExpr*), which is used to instruct Happer to perform certain operations for detecting packing behaviors. More specifically, *getArgValue* guides Happer to get a parameter value of a system function, and *setArgValue* instructs Happer to set a parameter value of a system function. In addition, *isFuncCalled* instructs Happer to find a system function from the resolved method invocations. Meanwhile, *dumpDex*, *isDexDiff*, and *isDexFragment* respectively guides Happer to retrieve the in-memory Dex file when an instruction is to be executed, find the difference between the Dex items and the retrieved Dex files, and determine whether the Dex items of the retrieved Dex file are loaded at dispersed memory regions. Moreover, *isInFile* and *isInMemRange* instructs Happer to check whether a string constant is included in the content of a system file and to examine whether a memory address is located in the memory region of a system library, respectively.

Example: Fig.9 presents the specification for the packing behavior AEU-1, which prevents the packed app from being run in emulators by inspecting the content of the system file that contains the fingerprint of emulators. Such behavior mainly consists of two operations.

First, the packer calls *fopen* defined in libc.so to access the system file /proc/tty/drivers, because the content of this file contains the names of the tty drivers created by emulators. Accordingly, since the first parameter of *fopen* stores the path of the accessed file, the specification tells Happer to retrieve the path string by using *getArgValue* (in Line 2), whose first parameter specifies the target system function and the second parameter provides the index of the parameter. Note that, in order to retrieve the parameter value, *getArgValue* internally instructs Happer to set a HBRK to the virtual address of *fopen*'s first instruction.

Second, since "goldfish" is the name of the tty driver created by emulators, the packer will call *strncmp* to find this string constant in the content of /proc/tty/drivers in order to know whether it is running in emulators. Based on this observation, the specification commands Happer to retrieve the value of the first and the second parameter of *strncmp* by using *getArgValue* (in Line 4 and Line 6). Similarly, to retrieve the parameter value, Happer will set a HBRK to the virtual

address of *strncmp*'s first instruction.

After getting the runtime parameter values of *fopen* and *strncmp*, Happer examines whether the retrieved file path equals "/proc/tty/drivers" and the string "goldfish" is involved in the string comparison (in Line 9). If both the conditions are satisfied, the AEU-1 packing behavior is found (in Line 10).

G. Identification of Packing Bahaviors

Tencent: Both Tencent-16 and Tencent-18 packers have DDL behaviors. However, Tencent-16 also employs ADG and DDM to prevent the packed app from being monitored and tracked by debuggers. ① Both of packers call *DexFile.loadDex* to load the protected Dex file (DDL-2). ② Tencent-16 forks a process and attaches it to itself to prevent the packed app from being analyzed by ptrace-based debuggers (ADG-2). ③ Tencent-16 erases the `header_item` of the protected Dex file, and restores it before the ART runtime calls *DexFile::<init>* (DDM-1).

Ali: Ali-16 adopts DDL and DDM to protect the packed app. ① It calls *DexFile.loadDex* to load the protected Dex file at runtime (DDL-2). ② The packer combines DDM-2 and DDM-3 to modify the loaded Dex file at runtime. In detail, it dynamically modifies the `annotations_off` field of `class_def_items` and adjusts the offset value stored in the `code_off` field of `encoded_method` items when the native library introduced by the packer is loaded for the first time.

Bangcle: The packing behaviors of Bangcle-18 includes ADG, DDL, and SLH. ① This packer forks a process and injects the created process to the packed app, which prevent the packed app from being attached by ptrace-based debuggers (ADG-2). ② The packer dynamically loads the protected Dex file through invoking *DexPathList.makePathElements* (DDL-3). ③ Bangcle-18 hooks several system library methods (e.g.,*ptrace*, *open*, *read*, *write*, and *close*) defined in `libart.so` and `libc.so` (SLH-1/2). For example, it hooks *ptrace* and inserts additional code to handle the `PTRACE_DETACH` parameter. Specifically, when *ptrace* is called with `PTRACE_DETACH`, the process of the packed app will be deliberately killed.

Ijiami: Both Ijiami-16 and Ijiami-18 apply ADG, AEU, ADI, TCK, and SLH to packing the app. ① They call *isDebuggerConnected* through the JNI reflection to detect debuggers (ADG-1). Moreover, these packers hook *AddListener* and *EnableMethodTracing* declared in the `Instrumentation` class to prevent them from being invoked by debuggers (ADG-4/5). In addition, the packers hook *__android_log_buf_write* to prevent the leakage of the log message (ADG-6). ② To detect whether the packed app is running in Qemu [28], Ijiami packers access the */proc/tty/drivers* file to check the presence of Qemu TTY driver (AEU-1). Moreover, values of specific system properties, especially for those introduced by Qemu, are examined (AEU-2/3). ③ These packers also implement ADI-1 to prevent the app from being unpacked by `ZjDroid`. ④ Furthermore, Ijiami packers deploy TCK-1 to protect the packed app. Specifically, the time limit for the target app to execute a specific native task is one second. Besides the behaviors for checking the running environment, Ijiami-16 adopts DDM-4 and Ijiami-18 uses DOM-1 respectively to protect the Dex file of the packed app. ⑤ Ijiami-16 fills `code_items` with valid bytecode at runtime. ⑥ Ijiami-18 dynamically adjusts the `dex_code_item_offset_` field of each related `ArtMethod` object to make it point to the dispersed `code_item`. ⑦ Note that behaviors ADG-4/5, DDM-4, and DOM-1 are implemented through hooking the related methods declared in system libraries `libart.so` and `liblog.so`, therefore, both of Ijiami packers have behaviors SLH-1/3.

Kiwi: DOM and DDF are realized by Kiwi-18 to protect the packed app. ① It inserts a method invocation to the static initialization method of each protected Java class. The method modifies the corresponding `mirror::Class` object via the native code (DOM-2). In detail, each `ArtMethod` object referred by the `methods_` field of the `mirror::Class` object is dynamically modified by the packer. Specifically, invalid values stored in the `access_flags` field and the `dex_code_item_offset_` field of each `ArtMethod` object are replaced with valid ones. ② Kiwi makes the offset value stored in the `dex_code_item_offset_` field of an `ArtMethod` object refers to a dispersed `code_item` element (DDF-3).

Testin: Testin-18 uses TCK, DDL, and JNT to protect the packed app. ① The packer leverages TCK-1 to prevent the target app from being debugged. More specifically, the predefined time limit is 10 seconds. ② Moreover, Testin-18 calls *DexPathList.makePathElements* to dynamically load the protected Dex file (DDL-3). ③ This packer uses the native code to reimplement each declared activity's lifecycle method, including *onCreate*, *onPause*, and *onResume* (JNT-1).