Parema: An Unpacking Framework for Demystifying VM-based Android Packers

Lei Xue
The Hong Kong Polytechnic University,
The Hong Kong Polytechnic University Shenzhen Research Institute,
China
leixue@comp.polyu.edu.hk

Yuxiao Yan
Xi'an Jiaotong Univerisity,
The Hong Kong Polytechnic University,
China
yanxibei@stu.xjtu.edu.cn

Luyi Yan
Muhui Jiang
Xiapu Luo*
The Hong Kong Polytechnic University,
China
{cslyan,csmjiang,csxluo}@comp.polyu.edu.hk

Dinghao Wu
Pennsylvania State University,
USA
dwu@ist.psu.edu

Yajin Zhou
Zhejiang University, Engineering Laboratory of Mobile Security of Zhejiang Province, China
yajin_zhou@zju.edu.cn

ABSTRACT

Android packers have been widely adopted by developers to protect apps from being plagiarized. Meanwhile, various unpacking tools unpack the apps through direct memory dumping. To defend against these off-the-shelf unpacking tools, packers start to adopt virtual machine (VM) based protection techniques, which replace the original Dalvik bytecode (DCode) with customized bytecode (PCode) in memory. This defeats the unpackers using memory dumping mechanisms. However, little is known about whether such packers can provide enough protection to Android apps. In this paper, we aim to shed light on these questions and take the first step towards demystifying the protections provided to the apps by the VM-based packers. We proposed novel program analysis techniques to investigate existing commercial VM-based packers including a learning phase and a deobfuscation phase.We aim at deobfuscating the VM-protected DCode in three scenarios, recovering original DCode or its semantics with training apps, and restoring the semantics without training apps. We also develop a prototype named `Parema` to automate much work of the deobfuscation procedure. By applying it to the online VM-based Android packers, we reveal that all evaluated packers do not provide adequate protection and could be compromised.

CCS CONCEPTS

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

KEYWORDS

App Protection, Obfuscation, Code Similarity

*Corresponding author

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464839

ACM Reference Format:
Lei Xue, Yuxiao Yan, Luyi Yan, Muhui Jiang, Xiapu Luo, Dinghao Wu, and Yajin Zhou. 2021. Parema: An Unpacking Framework for Demystifying VM-based Android Packers. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464839

1 INTRODUCTION

App packing services or packers are used to protect Android apps from being plagiarized and repackaged [20, 51]. Recent studies show that the apps packed by traditional protection mechanisms could be easily unpacked through dumping Dalvik bytecode (DCode) from memory [43, 47, 50], because the traditional app packing mechanisms follow a "write-then-execute" rule to release DCode in memory and then execute it by the virtual machine of Android system, denoted by `A-VM`.

To enhance protection capabilities, the latest Android packers adopt the VM-based protections that *never* release the DCode [52]. As shown in Fig. 1, during packing, the VM-based packer translates DCode to another customized type of bytecode, denoted by PCode, and embeds a customized virtual machine (`P-VM`) to interpret them when the packed/VM-protected app runs on device. The `P-VM` is typically implemented in a dynamically loadable native library and uses Java Native Interface (JNI) to interact with Android framework and runtime. When the VM-protected app runs on device, the PCode is dynamically released into memory, and the `P-VM` typically involves a decoding phase to parse the PCode and a dispatching phase to invoke the corresponding instruction handler, denoted as `PH`, to interpret the PCode. Since the corresponding original DCode of PCode are not required and removed from the packed apps, existing unpackers cannot find the original DCode in the memory, thus failing to unpack VM-protected apps. Therefore, the VM-based protection raises the bar of reverse-engineering packed apps, but little is known about whether such packers can be compromised and whether the apps protected by them can be unpacked.

Although there are some studies on VM-protected desktop programs [17, 19, 27, 33, 35, 41, 46], they cannot be applied to VM-based Android packers because they are not designed to recover bytecode

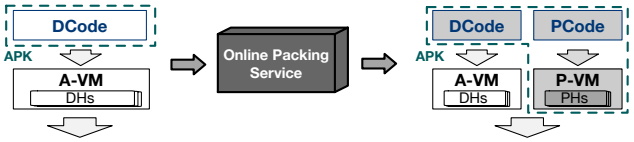


Figure 1: Overview of VM-based packing (DCode is the Dalvik bytecode interpreted by the DHs of A-VM; PCode is customized bytecode interpreted by the PHs of embedded P-VM).

or its semantics from the VM-protected apps. More precisely, existing studies aim at simplifying CFG (Control Flow Graph) [19, 46] or synthesizing the semantics of native code of the VM-protected programs [17, 27, 41]. However, the original code of VM-protected apps is bytecode, which is interpreted by A-VM with close interactions with other components (e.g., Android framework, libraries, etc.). Note that existing studies consider neither the semantics of bytecode nor cross-layer interactions.

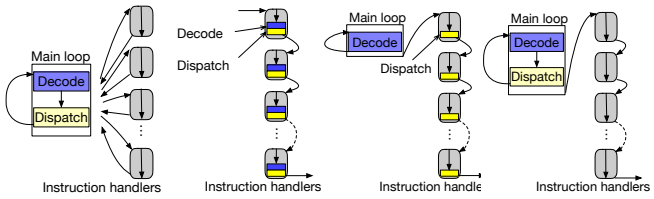
In this paper, we aim to demystify the newly emerging VM-based Android packers. Since a packer can design and implement its PCode and P-VM in fully customized ways, there comes a major challenge: *there is no information about PCode and its interpretation procedures.* To address this challenge, we propose a progressive solution to demystify the VM-based packers under three different scenarios. Since almost all existing commercial packing services are publicly accessible, we first investigate whether the VM-protected code (i.e., DCode) can be recovered with the help of the training apps, which are packed by the same version of packers as the target apps (D-1). Moreover, we investigate whether the semantics of the VM-protected DCode can be recovered with the assumption that the training apps and the target apps are packed by the different versions of packers(D-2). Furthermore, we investigate whether the semantics of the VM-protected DCode can be recovered if we cannot access any packer to build the training apps (D-3).

For *D-1*, we try to reverse-engineer the translation rules from DCode to PCode by learning the required information from the training apps. If the translation rules are reversible, the VM-protected DCode can be recovered through the learnt translation rules. For *D-2*, we construct the semantic features of all the PHs, which are represented by symbolic expressions, through analyzing the training apps. Then, we leverage them to recover the semantics of the PCode interpreted by the PHs of the packed apps for unpacking. For *D-3*, we construct the semantic features of all the DCode handlers (denoted by DHs) provided by the open-sourced A-VMs, and then recover the semantics of the PCode interpreted by the PHs through leveraging the semantic similarity between PHs and DHs.

To facilitate the investigation, we propose various novel unpacking techniques for VM-based Android packers and develop a prototype, named Parema, after addressing the challenges presented in §3. Applying Parema to the accessible VM-based packers, we reveal that these VM-based Android packers could still be compromised.

In summary, this paper makes the following contributions:

- We make a first step towards demystifying the newly emerging VM-based Android packers from three aspects (D-1/2/3) progressively. Our investigation on the accessible VM-based packers sheds light on their internals and capabilities of protecting Android apps.
- To assist the investigation, we propose a semantics-based solution to unpack VM-protected apps in three scenarios, and develop a prototype, named Parema, after tackling several challenging issues.



(a) Decode-dispatch (b) Indirect-threaded (c) Pre-decoding (d) Direct-threaded

Figure 2: The four generalized interpretation models, and the yellow, blue, and grey notes represent the decode, dispatch, and handler (i.e., PH) implementations, respectively.

Note that Parema can also recover the DCode protected by traditional methods and will be released at https://github.com/rewhy/parema.

- After applying Parema to 14 versions of 7 accessible popular commercial packers that claimed to adopt VM-based protections, we find that only four versions of two vendors actually adopted VM-based techniques. Moreover, the internals of their VM-based protections are reverse-engineered by Parema. Note that Parema also successfully unpacked other packers.

2 BACKGROUND

This section presents the necessary background knowledge of bytecode interpretation and a PoC (Proof-of-Concept) P-VM.

2.1 Interpretation of PCode

When a VM-protected app runs, the P-VM interprets each PCode instruction through three major steps, including fetching it from memory, decoding it, and dispatching proper handler (i.e., PH) to execute it. Although the packer providers can implement P-VM in an arbitrary manner, in practice, they typically realize P-VM following classic patterns considering the performance, development cost, and time to market [35, 37]. As shown in Fig. 2, they include decode-dispatch interpreter, indirect-threaded interpreter, pre-decoding interpreter, and direct-threaded interpreter.

The decode-dispatch interpreter (Fig. 2(a)) has a main loop where it interprets each instruction through three steps, namely *decode*, *dispatch* and *execution*. The indirect-threaded interpreter (Fig. 2(b)) addresses this issue by appending the decode-dispatch implementation to the end of each instruction handler. The pre-decoding interpreter (Fig. 2(c)) pre-decodes each instruction before dispatching and stores the decoding results (i.e., opcode and operands) in a structure. Consequently, the same instruction just needs to be decoded once, and the interpreter reuses the decoding results after the same instruction is interpreted. The direct-threaded interpreter (Fig. 2(d)) replaces the opcode with address of the corresponding instruction handler in the pre-decoding phase so that the interpreter can jump to the handler of next instruction at the end of current instruction handler.

In spite of the various implementations, each interpreter has an iterative procedure of decoding PCode and then dispatching PHs according to the decoding results. We regard this iterative procedure as the decode/dispatch procedure. Note that, since each PH of the indirect-threaded interpreter implements a decode/dispatch procedure (e.g., Fig. 2(b)), these iterative decode/dispatch procedures are represented by different nodes and have no loop pattern in the control flow even though they are implemented by the same code.

```
1 void vmpFunc(...) {
2   int a = 20, b = 25;
3   int s = a + b;
4   ...
5 }
```
Original implementation

```
1 native void vmpFunc(int m_id);
```
After VM-based obfuscation

Figure 3: The Java code disassembled from the DCode of a method before and after VM-protection.

```
 1 int vRegs[]; // The virtual register array
 2 int PHs[];   // The handler table
 3 int pwd = 0; // The additional (app-specific) decoding factor
 4 void PH_Const16(int vpc, int16_t PCode[]) {
 5   int16_t bytecode1 = PCode[vpc] ^ pwd;
 6   int16_t bytecode2 = PCode[vpc+1] ^ pwd;
 7   // Parse the operands
 8   int8_t  vR  = (bytecode1 & 0xff00) >> 8;
 9   int16_t value = bytecode2;
10   // Execute the semantics
11   vRegs[vR] = value;
12   vpc += 2; // Point the next instruction
13   // Decode  the next instruction
14   int8_t next_opcode = (PCode[vpc] ^ pwd) & 0xff;
15   // Dispatch handler for the next instruction
16   void (*pPH)(int, int16_t*);
17   pPH = (void(*)(int, int16_t*))PHs[next_opcode];
18   if (pPH != NULL):
19     pPH(vpc, PCode); // Invoke the target handler
20   return;
21 }
22 ...; // Other handlers
23 // Argument is the ID of the vm-protected method
24 // Argument is the ID of the target vm-protected method
25 void execute(int16_t m_id) {
26   // Initialize the virtual PC, registers and the handler table
27   int vpc = 0;
28   // Lookup the PCode of the target method according to m_id
29   int16_t *PCode = lookup_pcode(m_id);
30   init_virtual_registers(vRegs);
31   init_decoding_factor(&pwd); // pwd=0x0808 in this example
32   init_PCode_handlers(PHandlers);
33   // Decode the first instruction
34   int8_t opcode = (PCode[vpc] ^ pwd & 0xff00) >> 0x8;
35   // Dispatch PHandler for the first instruction
36   void (*pHandler)(int, int16_t*);
37   pHandler = (void(*)(int, int16_t*))PHandlers[opcode];
38   if (pHandler != NULL):
39     pHandler(vpc, PCode); // Invoke the target PHandler
40   return;
41 }
```

Figure 4: Code snippets of an indirect-threaded interpreter, including its entrance (i.e., *execute()*) and a PH (i.e., *PH_Const16()*).

2.2 A PoC P-VM of VM-based Packer

To illustrate the basic idea of VM-based protection, we design and implement a PoC P-VM following the basic packing mechanism of the VM-based packer in [51]. Fig. 3 presents a Java method *vmpFunc()* before and after being packed by this packer. The packing process consists of two steps: 1) turning the DCode to the intermediate bytecode by changing the opcodes following fixed rules, but the operands are unaltered; 2) converting the intermediate bytecode to PCode by XORing each of them with an app-specific parameter. Since the second step introduces the app-specific parameter, the mapping rules between PCode and DCode of different apps are variant. Moreover, the original method *vmpFunc()* is replaced by a native method that serves for context switch between P-VM and Android runtime. That is, the execution context goes into P-VM for interpreting the PCode when this VM-protected method is invoked, and returns to Android runtime from P-VM when it returns.

Meanwhile, we implement a P-VM (i.e., an indirect-threaded interpreter) to interpret the PCode. Fig. 4 shows its entry (i.e., *execute()*) and one PH (i.e., *PH_Const16()*) for demonstration. When the execution context switches to P-VM (Line 25), the interpreter first locates the PCode of the target VM-protected method according to the parameter `m_id` storing the method index. Precisely, the function *lookup_pcode()* (Line 29) is invoked to locate the memory region storing the PCode of the target method. After the virtual registers `vRegs` are initialized at Line 30, the app-specific parameter `pwd` is initialized by function *init_decoding_factor()* (e.g., `0x8080`) at Line 31. In Line 34, the virtual program register `vpc` points to the first PCode, which is decoded by XORing it with `pwd`, and then the proper PH is dispatched to interpret this PCode instructions according to the decoded opcode (Line 37). Afterward, the other PCode are iteratively decoded and interpreted in the similar way (i.e., Line 14 and 17).

To explain how PCode is executed by this P-VM (i.e., interpreter in Fig. 4), we take the first PCode instruction `<c208 081c>` of the VM-protected *vmpFunc()* in Fig. 3 (i.e., "`const/16 v0, #int 20`") as a concrete instance to illustrate the interpretation procedure. First, the bytecode `<c208>` is translated to the intermediate code `<ca00>` through XORing with `pwd` (Line 34 in Fig. 4), which is `<0x0808>` in this instance. Then, since the first byte `<ca>` is the opcode and indicates it is a `const16` instruction, the PH *PH_Const16* is dispatched and invoked to interpret this instruction in Line 34-39.

In *PH_Const16*, the operands are further decoded with `pwd` following the syntax of `const16` instruction at Line 5-9. In Line 11, the semantics of this instruction is interpreted with the two operands stored in the variables `vReg` and `value`. Afterward, the next instruction is decoded in Line 14, and a PH is dispatched and invoked to interpret it by the left code of *PH_Const16* (Line 16-20). Note that such interpretation procedures are iteratively executed until a `return` instruction is interpreted.

CFG/SCFG: As shown in Fig. 4, since the PHs implements the same decode/dispatch procedures (e.g., Line 12-19 and Line 33-39) repeatedly, they are represented by different nodes in CFG. Consequently, there is no loop pattern in the CFG although their implementations are same. Since the loop patterns of decode/dispatch procedures are usually used to handle desktop VM-based obfuscation [35], we propose a novel data structure SCFG (Symbolic Control Flow Graph, in §4.1.2) to represent the code for the sake of identifying the aforementioned loop patterns. SCFG uses one node to represent all the code blocks implementing the same semantics/logic, and it is different from CFG, of which each node is one actual code block stored in a concrete memory region. Hence, the iterative decode/dispatch procedures implemented with the same code in different PHs are represented by the same nodes in SCFG and have the loop pattern.

Table 1 summarizes the major abbreviations used in this paper as well as their corresponding representations.

3 OVERVIEW

As shown in Fig. 5, our unpacking solution consists of a *learning phase* and a *deobfuscation phase* represented by white and gray rectangles, respectively. The two white rectangles mean that we can learn the information from either customized training apps or the AOSPs (Android Open Source Projects) for deobfuscation.

In *learning phase*, we learn the necessary information through two ways. ***L1)*** We use customized apps (i.e., training apps) to explore the VM-based packers for collecting necessary information (i.e., reversible DCode/PCode translation rules and semantic features of the PHs) of P-VMs directly (§4.1); ***L2)*** We learn the semantic

Table 1: Notations of the major information and transfer rules leveraged during investigation.

Name	Representations
DCode	The Dalvik bytecode in the Android apps before being packed.
PCode	The customized bytecode tranlated from Dalvik bytecode during being packed.
A-VM	The Android VM for executing DCode.
P-VM	The customized VM for interpreting PCode in the packed apps.
DH	The instruction handlers of A-VM for interpreting DCode.
PH	The instruction handlers of P-VM for interpreting PCode.
PAM	PCode addressing mechanism. It is used by P-VM to locate the PCode of the VM-protected method.
P2PH	The mapping from PCode to PHandler.
PH2D	The mapping from PHandlers to DCode instructions.
SCFG	The symbolic control flow graph, containing more semantic information and more robust compared to CFG.
O_{ph}/S_{ph}	The offset/semantics features of PHs.
O_{dh}/S_{dh}	The offset/semantics features of DHs.

features of DCode interpretation procedures (i.e., DHs) of the A-VMs (i.e., AOSPs), and then infer the semantics of interpreted PCode through the semantic similarity analysis between DHs and PHs for deobfuscation (§4.2).

The *deobfuscation phase* aims at recovering the original DCode or their semantics of a VM-protected app by leveraging the information obtained in learning phase.

Scope and Assumptions: It is difficult, if not impossible, to unpack apps protected by an arbitrary VM-based packer without any information due to both the freely defined PCode and implemented P-VM. In this first study towards demystifying VM-based Android packers, we focus on the publicly available packers (e.g., online providers [14–16, 26, 32, 38]), which have already been widely used by both benign and malicious apps [21].

We demystify VM-based packers in three different scenarios (i.e., D-1, D-2, and D-3). For D-1 and D-2, we assume that the packers are accessible. This assumption is rational for Android apps because the publicly available packers (e.g., online providers) have been widely used by many apps. With this assumption, we can restore the VM-protected DCode in D-1 and recover the semantics of VM-protected DCode in D-2. More importantly, we are the first to reveal that such widely-used packers cannot provide sufficient protection.

For D-3, since AOSPs are open-source, we do not assume the packers are accessible, and our approach can still recover the semantics of VM-protected DCode and restore the DCode protected by traditional methods. It is worth noting that we conduct the first study on VM-based packers for Android apps and existing studies on deobfuscating VM-based desktop programs cannot recover the detailed semantics of VM-protected code like ours.

We assume that the P-VM is implemented with a register-based interpreter and follows one of the four interpretation models in Fig. 2. Moreover, one type of DCode instruction is translated to a fixed number of PCode instructions. These assumptions are rational because they are the common practices for implementing interpreter [36], and the online packers follow them for the consideration of performance, development cost, and time to market. For instance, the A-VMs of all AOSPs adopt the register-based architecture, and all VM-based packers examined in this study meet these

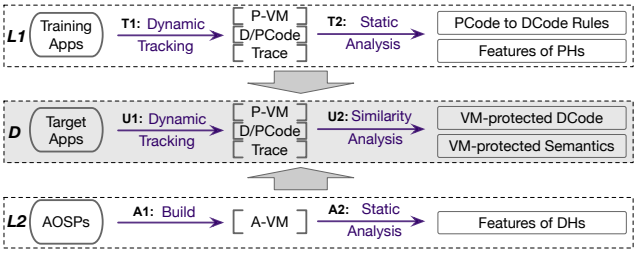


Figure 5: An overview of our solution. The *training phases* (i.e., white rectangles) take in the training apps or the AOSPs. The *deobfuscation phase* (i.e., grey rectangle) recovers the VM-protected DCode or their semantics by leveraging the learnt information.

assumptions. As another example, Sharif et al. [35] found that the interpreters of VM-based desktop packers follow the decode-dispatch interpretation model (Fig. 2(a)).

Challenges: As show in Fig. 5, to examine VM-based packers, we need to first locate the PCode and PHs of the packed apps as well as reverse-engineer DCode/PCode translation rules. Then, we identify the PHs' semantic features in learning phase (i.e., ***L1***) and recover the VM-protected DCode or their semantics in *deobfuscation* phase. However, even if P-VMs follow certain interpretation patterns, they are close-sourced and diverse, thus introducing three challenges.

C-1) Locating PCode and PHs: Since packers dynamically release PCode to memory along with a mass of other data, it is challenging to locate the PCode. Also, as shown in Fig. 2, the PHs can be dispatched with different implementations, making it difficult to locate the PHs. To address these issues, we propose the novel SCFG to uniformize the various decode/dispatch models in §4.1.2.

C-2) Determining the decode/dispatch rules of PHs: Besides the diverse decode/dispatch implementations, the P-VM can also involve app-specific factors to increase the strength of obfuscation. To address this issue, we conduct differential symbolic expression analysis to reverse-engineer the decode/dispatch procedure in §4.1.2.

C-3) Recognizing semantics of VM-protected DCode: Since the PCode are freely defined by the packer providers and the apps are packed online, we have no pre-knowledge of PCode. To overcome this issue, we propose to learn the semantic features of PHs from training apps or that of DHs from AOSPs, and then use them to infer the semantics of VM-protected DCode in §4.3.2 and §4.3.3.

Investigation Flow: As shown in Fig. 5, we investigate D-1/D-2 with the information learnt from training apps and D-3 using the information learnt from A-VMs. The results are used to demystify the packers claiming to adopting VM-based protections.

D-1: Due to security considerations, the packers usually adopt irreversible rules to convert DCode to PCode when the apps are packed online. Hence, we explore the VM-based packers by reverse-engineering the PAM (PCode address mechanism), P2PH (decode/dispatch procedure), and then build PH2D (i.d., the mapping relationship from PHs to DCode) by analyzing the training apps in *learning phase*. In *deobfuscation phase*, we investigate D-1 through locating PCode, determining PHs for interpreting the PCode, and recovering DCode from PHs by leveraging the learnt knowledge.

D-2: Since the recovery of DCode depends on the learnt PAM, P2PH, and PH2D, the training apps and the target apps need to be

packed by the same version of a packer. However, packers are updated frequently, and thus we also explore whether the semantics of the interpreted PCode can be recovered with the knowledge learnt from the training apps packed by the different versions of a packer. We generate the semantic features of the PHs by conducting symbolic analysis of the training apps, and then apply semantic similarity analysis to recognize the semantics of the PCode interpreted in the target apps during deobfuscation.

D-3: We further explore whether the semantics of interpreted PCode can be recovered if we cannot create training apps by packing customized apps with the VM-based packers. Specifically, we generate the semantic features of the DHs by conducting symbolic analysis of the A-VMs from various AOSPs, and then recognize the semantics of the interpreted PCode in the target apps through semantic similarity analysis in *deobfuscation phase*.

4 INVESTIGATION

This section details our methodology for investigating the VM-based Android packers involving *learning* and *deobfuscation* phases. In *learning phase*, we first prepare training apps by letting their functions to be protected contain all possible DCode instructions and then upload them to online VM-based packers. As shown in Fig. 5, after retrieving the VM-protected ones, we conduct dynamic app tracking (Step **T1**) to collect their execution traces and other information (i.e., P-VM and PCode) for profiling the P-VM (Step **T2**) in §4.1. We also build various AOSPs (Step **A1**) and carry out symbolic analysis of their A-VMs (i.e., libart.so and libdvm.so) to extract semantic features from DHs (Step **A2**) in §4.2. In *deobfuscation phase*, for each packed app, we first locate its P-VM and PCode, and log the execution trace during dynamic tracking (Step **U1**). Then, we leverage the knowledge learnt from the training apps or the A-VMs to recover the VM-protected DCode or the executed semantics, which represents the semantic information of executed PCode (Step **U2**) in §4.3.1 and §4.3.2 respectively.

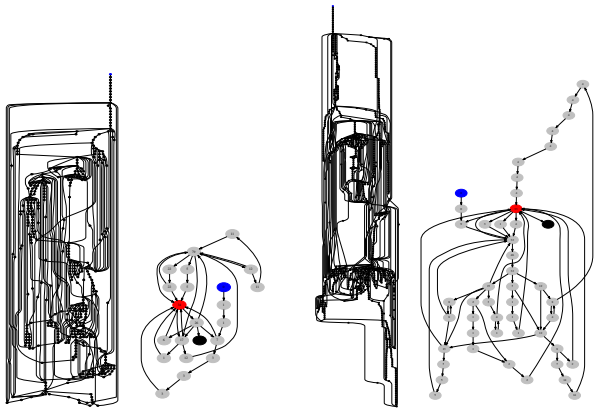
To automate the most tedious analysis work in our methodology, we implement the framework Parema based on the dynamic binary instrumentation framework Valgrind [9] and the symbolic analysis engine Angr [10], both of which use the VEX IR as intermediate representations. Although we focus on deobfuscating the VM-protected DCode in this paper, Parema also unpacks the traditionally protected DCode during our investigation.

4.1 Learning with Training Apps

We analyze training apps to learn the necessary information of PAM, P2PH, PH2D, and the semantic features of PHs for deobfuscation.

*4.1.1 Dynamic Tracking (**T1**).* We first run the training apps on a real device and record the executed VEX IR with concrete data as well as the invoked Android framework APIs and library (libart.so and libc.so) functions in trace files. Meanwhile, we dump the customized native code and data, including the implementation of P-VM and PCode, respectively, for further static analysis.

Since the native code requires executable permission, we locate the customized native code through looking for the memory data that have executable permission and is not loaded from system libraries. Due to security consideration, the packer dynamically decrypts and releases both DCode protected by traditional methods



(a) CFG (Qihoo) (b) SCFG (Qihoo) (c) CFG (Baidu) (d) SCFG (Baidu)

Figure 6: The CFG and the corresponding SCFG produced by the interpreter of the P-VM from two VM-based packers, Qihoo and Baidu, which adopt decode-dispatch and indirect-threaded interpreters, respectively. The red node in the SCFG is the entry of each decode/dispatch procedure. The blue and black nodes are the entry and exit nodes of the interpreter, respectively.

and the PCode related data into memory, which are then interpreted by A-VM and P-VM, respectively. Therefore, we dump the data dynamically written into memory by customized native code (i.e., native code of packer) as the potential DCode and PCode for further analysis. Note that, without packing, the DCode related data is stored in the Dex files, which are loaded into memory when the app starts and then interpreted by A-VM.

*4.1.2 Static Analysis (**T2**).* Since P-VM continuously fetches and executes each PCode instruction, such behavior will result in an iterative procedure in the execution trace. As shown in Fig. 2, the iterative procedures have various pattens (e.g., loop) in CFGs due to different implementations. Hence, it is non-trivial to identify such iterative procedures from the execution trace.

To address this issue, we propose a novel structure named SCFG to characterize the iterative procedures of different interpretation implementations (i.e., Fig. 2) in a uniform format, and then locate the PCode and their corresponding PHs by analyzing SCFG. In CFG, each node represents a basic code block at a specific memory region, and hence even if two code blocks have the same implementation, they are denoted by two different nodes. To build SCFG, we convert each code block into a set of symbolic expressions through symbolic execution, and then the blocks producing similar symbolic expressions are represented by the same node in SCFG. Thus, the code blocks represented by the same node in SCFG have the same functionality although they are implemented separately and located at different memory regions. Therefore, the decode/dispatch procedures of the P-VM still have iterative/loop patterns in SCFGs.

The CFG and SCFG shown in Fig. 6 illustrate how the VM-based packers (i.e., Qihoo and Baidu) interpret the same VM-protected method by contrast. We can find the SCFG is much conciser than the CFG and the iterative decode/dispatch procedure has obvious loop pattern starting from red node in the SCFG, which is not found in the CFG.

Locating PCode and PHs: Since the decode/dispatch procedure involves PCode and PHs as input and output respectively, we identify the routines that implement this procedure by exploiting the loop pattern in SCFG through the following five steps:

I) *Determining Candidate PCode*: We selects the data loaded from memory as the candidate PCode because PCode is fetched by P-VM from the memory. The IR operation `Load` is used to load data from memory in execution trace and its input is the source address.

II) *Generating Symbolic Expressions*: We leverage symbolic expressions to characterize the relationship between each candidate PCode and the related IR statements. In particular, we represent each candidate PCode as a symbolic input and perform symbolic execution on traced IR statements with the other concrete inputs. Then every variable related to the candidate PCode is represented by a symbolic expression containing the symbolic inputs.

During symbolic expression generation, a symbolic expression is emitted when any of the following scenarios occur.

• S-①: A store statement (`IR_Store`/`IR_StoreG`) is executed and its target address can be represented by a symbolic expression containing a candidate PCode. In this scenario, the target address has a potential relationship with the PCode.

• S-②: The JNI reflection functions are invoked and the parameters can be represented by symbolic expressions. The JNI reflection functions refer to the functions provided by Android runtime (in libart.so), and P-VM needs to call these functions for invoking the target methods when interpreting method invocation instructions.

• S-③: Either a conditional branch (i.e., `IR_Exit`) or an indirect branch (i.e., `IR_Goto`) is executed, and meanwhile the condition or the target address can be represented by a symbolic expression. In this scenario, P-VM dispatches the proper PHs according to the PCode's opcode (i.e., the symbolic variable).

III) We use SCFG to represent the control flow of the interpreter for the ease of recognizing the decode/dispatch loop as well as its input and outcome, which correspond to the PCode and PHs. Note that SCFG has three major advantages over CFG, a) all its nodes have a potential relationship with the PCode since the symbolic expressions of its nodes are generated from the candidate PCode; b) it has more semantic information than CFG since each node in SCFG is one or more symbolic expressions representing a process on the candidate PCode; c) the structure does not change due to the repeated implementations of the same functionalities/logic (i.e., the decode/dispatch procedure) because the nodes of SCFG are created according to the symbolic expressions instead of memory addresses of the code blocks.

IV) To identify the iterative decode/dispatch procedure, we use the loop detection approach proposed in [13] to find the loops in SCFG and employ one heuristic to filter out irrelevant ones. Since a decode/dispatch loop starts from fetching a PCode from the memory, each instance of the entry node of the loop should contain a symbolic input corresponding to a new PCode. Moreover, since the outputs (i.e., PHs) of decode/dispatch procedure depend on the input (i.e., PCode), the branch targets of the exit blocks depend on the symbolic input. In this step, manual effort may be involved to determine the actual decode/dispatch procedure if multiple loops are found. Such effect is just required once for each packer.

V) After identifying the decode/dispatch procedure as well as its entry and exit nodes in SCFG, we determine the opcodes of the

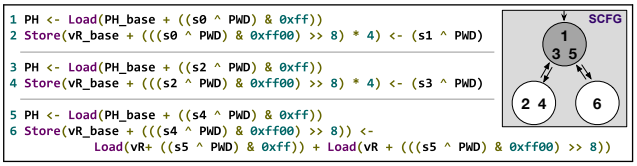


Figure 7: The symbolic expressions and SCFG generated by three PCode instructions (i.e., "`const/16`", "`const/16`" and "`add-int`"), and grey node is the entry of iterative decode/dispatch procedure.

PCode that are fetched by the decode/dispatch procedure. Since P-VM interprets each PCode instruction by fetching the opcode that actually determines the PH, the entry node of decode/dispatch loop loads the PCode opcode. Thus, we first identify the PCode opcode according to the symbolic inputs initialized by the entry node of the decode/dispatch loop, such as the symbolic variables `s0`, `s2` and `s4` in the entry node of the SCFG shown in Fig. 7. Afterwards, since these PCode opcodes are in the real PCode region, we regard the continuous memory regions that store these PCode opcodes as the PCode regions. All PCode, including opcodes and operands, is usually stored in the PCode regions. Consequently, the branch targets of the decode/dispatch loop are the starting addresses of the PHs that handle the PCode.

Abstracting Decode/dispatch Procedure (P2PH): The decode/dispatch procedure may involve app-specific parameters for increasing protection strength. For instance, in two apps packed by the same VM-based packer, the PCode instructions could be decoded and dispatched in a different manner due to the involved app-specific parameters. Hence, to comprehensively understand the decode/dispatch procedure, we further abstract this procedure by identifying such app-specific parameters using differential symbolic analysis.

We achieve this goal by examining the symbolic expressions generated by the decode/dispatch procedures of different training apps. Specifically, we compare the concrete values of the symbolic expressions generated by the decode/dispatch procedures of different VM-protected apps, and regard the concrete values that are not fixed as the app-specific parameters.

Since the decode/dispatch procedures in different apps hardened by the same packer are usually same but involve some variants due to app-specific parameters, the P2PH learnt from training apps is represented by one or more symbolic expressions that take in the PCode and app-specific parameters and then output the target PHs. These symbolic expressions can be applied to other apps protected by the same packer. We emphasize that such analysis only needs to be conducted *once* for a VM-based packer.

Identifying Deobfuscation Rules (PH2D): To recover the VM-protected DCode or the executed semantics, we design two kinds of features, namely *offset* and *symbolic/semantic* features, denoted by O_{ph} and S_{ph}, for recovering DCode and semantics, separately. O_{ph} refers to the offset address of each PH, and S_{ph} includes the symbolic expressions produced by each PH with PCode as symbolic inputs/variables. For instance, when interpreting a method invocation instruction (i.e., `invoke-`$*$), the corresponding PH needs to invoke the target methods through JNI reflection functions [8] according to the index of the target method. Hence, the symbolic feature also includes such JNI reflection functions.

After building the semantic feature of each PH, we determine the mapping rules (i.e., PH2D) from the O_{ph} and S_{ph} of PHs to the DCode instructions and the executed semantics, respectively. Moreover, when interpreting an instruction, the VM dispatches the target handler according to its opcode and then the handler executes specific semantics with its operands, and hence we also identify both the opcode and the operands that comprise each PCode instruction by symbolic analysis of the whole interpretation trace of it. Since the purpose of O_{ph} is to recover the complete original DCode including both opcodes and operands, we build the PH2D from O_{ph} to DCode, which includes the recovery rules from the PCode operands executed by PHs to the DCode operands.

Consequently, in deobfuscation phase, we can recover the DCode opcode according to the dispatched PH as well as the DCode operands from the PCode opcodes used by the PH. When S_{ph} is used, we focus on recovering the executed semantics instead of the original DCode. We emphasize that, in the deobfuscation phase, if we aim to recover all VM-protected DCode using O_{ph}, P2PH, and PH2D, the target app and the training apps need to be packed by the packer of the same version, whereas recovering the executed semantics using S_{ph} has no such requirement.

4.2 Learning with Android VMs

We further investigate whether the semantics of the interpreted PCode can be recovered without training apps (i.e., D-3). Since DCode is translated to PCode during packing, they have the same semantics and are interpreted by the PHs and DHs, respectively. Thus, in learning phase (i.e., Fig. 5), we learn the semantic features of DHs (i.e., S_{dh}) by the analysis of various A-VMs.

*4.2.1 Building A-VMs (**A1**).* Although the A-VMs of all Android systems interpret the same DCode/semantics, their implementations are diverse in different Android systems (i.e., AOSPs). Hence, to let S_{dh} contain only the crucial semantic features, we build the AOSPs of seven Android systems (4.4-10.0) for further analysis.

*4.2.2 Static Analysis (**A2**).* We generate S_{dh} by analyzing the A-VMs of various versions and use $S_{dh}[d]$ to denotes the DH interpreting the DCode instruction d, such as `invoke`, `move`, `compare`, and `if`. We summarize the supported semantics in Table 2, including 22 types of semantics belonging to 8 categories and all DCode instructions. Note that, the handlers executing the same type of semantics have similar implementations and functionalities because their target PCode/DCode instructions stand for the same operations with only different operand types (e.g., `int16` and `int32`).

To extract the semantic features of DHs (i.e., S_{dh}), we carry out similar symbolic analysis on A-VM as that on P-VM. More specifically, we first identify the binary instructions of each DH by analyzing the library libart.so and symbolically execute it by specifying the input DCode as symbolic inputs. Then, we obtain all symbolic expressions related to the DCode. Since we build S_{dh} through analyzing various A-VMs, for each DH, we get multiple sets of symbolic expressions and each set contains the semantic expressions generated by the DH of a specific version of A-VM. To further simplify the semantic features, we further purify each S_{dh} by removing the symbolic expressions shared by the minority (less than half) expression sets of this DH. We also simplify all symbolic expressions using Z3 solver [12].

Table 2: The semantic features extracted from the A-VMs.

Category	Semantics/Operations	Description
Data Transfer	`move* except move-exception/result*`	Move the contents of one register to another
	`move-result* move-exception`	Move the result of the most recent `invoke-kind` or the `just-caught` exception into the indicated register
	`const* except const-method-* const-class/string*`	Move the given literal value into the specified register
Branching	`goto*`	Unconditionally jump to the indicated instruction
	`if-*`	Branch to the given destination if the given two registers' values compare as specified
	`*switch`	Jump to a new block based on the value in the given register or fall through to the next instruction if there is no match
	`return*`	Return from a method
Comparison	`cmp-*`	Perform the indicated floating point or long comparison
Number ops	`*sh*-int* *sh*-long*`	Perform the indicated binary shift op on the indicated register and another register or a literal value, storing the result in the destination register
	`neg-int, neg-long`	Unary twos-complement
	`not-int, not-long`	Unary ones-complement
	`*-int*, *-long*`	Perform the indicated op on the indicated register and another register or a literal value, storing the result in the destination register
	`*-float*, *-double*`	Perform the indicated floating point op on the indicated register and another register, storing the result in the destination register
	neg-float, neg-double	Floating point negation
	`*-to-*`	Data type conversion
Array ops	`*new-array*`	Construct a new array of the indicated type and size
	`fill-array-data aget*, aput*`	Access an array
Invocation	`invoke-*`	Call the indicated method
Field access	`iget*, iput*`	Access the identified instance field
	`sget*, sput*`	Access the identified static field
Exception process	`throw`	Throw the indicated exception
	`monitor-*`, etc.	Other semantics that are not closely related to data flow and control flow

4.3 Deobfuscating VM-protected Apps

This section will detail how we deobfuscate the VM-protected DCode based on the knowledge learnt from training apps or AOSPs in three scenarios. As shown in Fig. 5, in *deobfuscation phase*, we first dynamically trace the target app to locate and dump the native implementation of the P-VM, the memory data containing DCode and PCode, as well as the execution trace (Step **U1**) through the same way described in §4.1.1, and then analyze these traces (Step **U2**) to achieve the deobfuscation purposes.

4.3.1 D-1: Recovering DCode. To recover the VM-protected DCode, we use the learnt P2PH and PH2D to determine the PH for interpreting every PCode instruction, and recover the original DCode instruction from each identified PH, respectively. When recovering DCode, we use the offset features O_{ph} to represent the PHs. If the recovered DCode is same as the original one, the VM-protected DCode can be recovered with training apps (i.e., D-1).

4.3.2 D-2: Recovering Semantics with Training Apps. To recover the semantics executed by P-VM, we first use the learnt S_{ph} (§4.1)

to recover the semantics of the interpreted PCode by analyzing both the native code of P-VM and the execution trace (Step **U2**). More precisely, we first extract the semantic features $\tilde{S}_{ph}$ of the PHs in the P-VM of target app through symbolic analysis (i.e., $\tilde{S}_{ph}$ denotes the features of PHs to be recognized). For every $\tilde{S}_{ph}[i]$ of the PH indexed by i (i.e., $\tilde{ph}_i$), we calculate the similarities between it and all $S_{ph}[d]$, where d represent a specific type of supported semantics in Table 2. We regard the semantics executed by $\tilde{ph}_i$ as the same as that of the $S_{ph}[d]$ with the largest similarity to $\tilde{S}_{ph}[i]$. Currently, Parema calculates the semantics similarity between $\tilde{S}_{ph}[i]$ and $S_{ph}[d]$ using the Jaccard similarity coefficient [31] as $Sim(\tilde{S}_{ph}[i], S_{ph}[d]) = Jaccard(\tilde{S}_{ph}[i], S_{ph}[d])$.

After recognizing the semantics of all PHs, we continue to determine the executed semantics according to the invoked PHs in the execution trace. During dynamic tracking (Step **U1**), we also track the invoked library (i.e., libart.so and libc.so) functions and the framework APIs implemented in boot.oat , thus the methods invoked by the invoke instructions are further identified according to the execution trace. Finally, we investigate D-2 by comparing the recovered semantics and the semantics of original DCode. If they are same, D-2 has a positive answer, otherwise a negative one.

4.3.3 D-3: Recovering Semantics with DH features. We further explore recovering the semantics executed by P-VM based on the learnt S_{dh} (in §4.2) for D-3. More precisely, we use the same way described in §4.3.2 but with S_{dh} learnt from A-VMs instead of S_{ph} to recover the semantics of interpreted PCode by analyzing the native code of P-VM and the execution trace (Step **U2**). If the recovered semantics are same as that of the corresponding original DCode, the semantics of VM-protected DCode (i.e., PCode) can be recovered without training apps (i.e., D-3).

5 EVALUATION

We implement the investigation framework Parema composed of a dynamic tracking module and a static analysis module with around 11.5k lines of C/C++ code and 6.8k lines of Python script. Parema also adopts the mechanism similar to PackerGrind [43] to dynamically collect the DCode protected by traditional methods in memory. We evaluate Parema and investigate the VM-based Android packers by answering the following research questions (**Q1-3**).

• **RQ1:** Can Parema effectively locate the PCode, P-VM/PHs, and learn the required knowledge (i.e., P2PH, PH2D, and O_{ph}) from training apps for recovering the DCode of VM-protected apps?

• **RQ2:** Can Parema extract the semantic features of the PHs and DHs (i.e., S_{ph} and S_{dh}), which are implemented in the VM-based packers and the AOSPs, respectively, to recover the semantics of the interpreted PCode of the target VM-protected apps?

• **RQ3:** What are the results of using Parema to investigate the existing VM-based packers for Android apps?

5.1 Android Packers and Data Sets

Packers: We investigate seven public accessible commercial packer providers, including Ijiami [26], Bangcle [16], Baidu [15], Qihoo [32], Ali [14], Tencent [38], and APKProtect [6], and use them to pack our customized apps. If the protected DCode of *all* invoked methods is interpreted by A-VM, the packer does not conduct VM-based protection. Although all these packers except APKProtect claim to adopt VM-based protection, we find only Baidu and Qihoo actually adopt VM-based protection after checking. They employ the hybrid protection that protects critical methods using VM-based mechanism and other methods with traditional mechanisms.

Data sets: We prepare three data sets, i.e., TSetA, VSetA, and VSetB for investigation. TSetA contains 70 customized apps that cover as many DCode instructions as possible and it is used for training. VSetA and VSetB are two testing app sets, and they are generated by packing 30 apps, which are randomly downloaded from the open-source app repository F-Droid [2], with different versions of packers. Specifically, the apps in VSetA and TSetA are packed using the *same version* of packers, whereas apps in VSetB are packed with updated packers (i.e., *different versions* of the packers). All data sets and an illustrative/motivating example are available at [11].

Consequently, for D-1, we unpack the apps in VSetA using the information learned from TSetA. For D-2, we unpack the apps in VSetB with the help of TSetA. For D-3, we unpack VSetA and VSetB without using training apps. It is worth noting that the packers of two different versions usually adopt similar VM-protection technologies but different code translation rules. For example, we found that both versions of Baidu packers were implemented using the indirect-threaded interpreter (Fig. 2(b)), and the two versions of Qihoo packers adopted decode-dispatch interpreter (Fig. 2(a)). Moreover, the two versions of both Baidu and Qihoo packers used different code translation rules for packing.

Table 3: The comparison between CFG and SCFG generated during interpreting the *onCreate* methods of MainActivitys on #node|#edge.

App	#PCode instruction	Qihoo		Baidu	
		CFG	SCFG	CFG	SCFG
dowhile	1712	407|533	26| 43	970|1139	68| 98
enumeration	95	538|699	25| 42	926|1127	69|105
regex3	65	485|635	26| 42	986|1203	65| 90
floatmod	26	411|535	31| 49	820| 955	35| 46
createwidget4	13	348|439	18| 29	700| 784	32| 41

5.2 RQ1: DCode Recovery

To locate the PCode and the PHs of P-VM, we dynamically track the training apps and statically analyze their execution traces.

5.2.1 Constructing SCFG. We first construct the SCFGs of the packed apps using the approach presented in §4.1.2. Table 3 summarizes the CFG and SCFG information of five randomly selected apps. It shows that the SCFGs include much fewer nodes and edges than the CFGs, and thus they ease the identification of the routines that implement the decode/dispatch procedure and the localization of the PCode and PHs.

5.2.2 Locating Real PCode. After identifying the entry nodes in the SCFGs using the approach described in §4.1.2, we first identify the entry nodes that are represented by the symbolic expressions "*cons=((((((0x9090^p)≪0x18)≫0x18)-0x1)≥0xfe)*" and "*dest=(load(((0x6ce2e60+((0xff & p)≪0x2))+0x4))|0x1)*" in the training apps packed by Qihoo

Table 4: The symbolic expressions of entry nodes generated by various apps.

App	Qihoo		Baidu	
	Symbolic Expression	Factor (i.e., f_{qihoo})	Symbolic Expression	Factor (i.e., f_{baidu})
com.uberspot.a2048	*((((((0xe6e6^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	*0xe6e6*	*(load(((0x6c84958+((p&0xff)≪0x2))+0x4))|0x1)*	*0x6c84958*
fr.asterope	*((((((0x1010^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	*0x1010*	*(load(((0x6ccdd98+((p&0xff)≪0x2))+0x4))|0x1)*	*0x6ccdd98*
net.mathdoku.holoken	*((((((0x0a0a^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	*0x0a0a*	*(load(((0x6ccfe58+((p&0xff)≪0x2))+0x4))|0x1)*	*0x6ccfe58*
net.tevp.postcode	*((((((0x0c0c^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	*0x0c0c*	*(load(((0x6ccdd28+((p&0xff)≪0x2))+0x4))|0x1)*	*0x6ccdd28*
org.ligi.passandroid	*((((((0xeaea^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	*0xeaea*	*(load(((0x6cd0b68+((p&0xff)≪0x2))+0x4))|0x1)*	*0x6cd0b68*
Summary	*((((((*f_{qihoo}*^p)≪0x18)≫0x18)-0x1)⩾0xfe)*	f_{qihoo}	*(load(((*f_{baidu}*+((p&0xff)≪0x2))+0x4))|0x1)*	f_{baidu}

and Baidu, respectively. Then, we determine the actual PCode opcodes (i.e., the concrete inputs of the symbolic variable p in the expressions). Afterwards, we locate the PCode regions of the VM-protected methods according to the locations of these opcodes.

5.2.3 Generalizing Decode/Dispatch Procedure. We identify the decode/dispatch procedures by analyzing the symbolic expressions in the SCFGs of all training apps packed by the same packer. For instance, Table 4 lists the symbolic expressions of the entry points in five training samples, and f_{qihoo} and f_{baidu} are the variational parameters involved in the decode/dispatch procedures. For Qihoo, f_{qihoo} is an integer involved in decoding the PCode and it is not fixed in different packed apps. For Baidu, f_{baidu} is the base address of a jump table, which changes in different runs. Hence we treat f_{qihoo} and f_{baidu} as the app-specific parameters.

Afterwards, we recognize the opcode of PCode by analyzing the symbolic expressions. As shown in Table 4, since the subexpression "*(((*f_{qihoo}*^p)≪0x18)≫0x18)*" within the symbolic expression of Qihoo is to obtain the low 8 bits of the PCode (i.e., the result of "f_{qihoo}^p"), such 8 bits refer to the opcode. Similarly, for Baidu, since the subexpression "*(*f_{baidu} *& 0xff)*" is used to obtain the low 8 bits of the PCode, the opcode is in the low 8 bits. In summary, the Qihoo and Baidu packers decode the opcode op from the input PCode p according to the changed concrete values in the expressions "*op=(((*f_{qihoo}*^p)≪0x18)≫0x18)*" and "*op=(*f_{baidu} *& 0xff)*", respectively.

We further identify the dispatching processes of the P-VMs of Qihoo and Baidu according to their symbolic expressions, namely "*((load((j+((op-0x1)≪0x2)))+j)|0x1)*" and "*(load((((j+(op≪0x2))+0x4))|0x1)*", which represent the addresses of the invoked PHs. The variable *op* is the PCode opcode and *j* is the base address of the jump table, whose concrete value is identified from execution trace in both learning and deobfuscation phases. Thus we get the complete information about the decode/dispatch procedure (P2PH) and the routines that realize this procedure by combining the decoding and dispatching processes. These information will be leveraged to determine the PHs according to the PCode (i.e., p) and the concrete values of the app-specific parameters (i.e., f_{qihoo}, f_{baidu}). After that, the offset addresses of PHs (i.e., O_{ph}) is identified and used to recover the DCode for deobfuscation.

5.2.4 DCode Recovery. We run `Parema` to recover both VM-protected DCode and DCode protected by traditional methods of the packed apps in `VSetA` using the knowledge learnt from `TSetA`. To compare `Parema` with the off-the-shelf unpackers, we also use three popular off-the-shelf unpackers listed in the first column of Table 5 to unpack the same apps. Table 5 also shows the unpacking results, where "Partial Dex File" means that the unpacking results contain only the traditionally protected DCode and "VM-protected Method" denotes that the VM-protected DCode is also recovered. Moreover, "applicable" in Table 5 means that the semantics or the corresponding original DCode of the PCode in the VM-protected apps are correctly recovered, and we determine the correctness by comparing the recovered semantics with the corresponding DCode before being VM-protected. It shows that only `Parema` can recover the VM-protected DCode.

Table 5: Comparison between Parema and existing tools (✓ represents "applicable" and ✗ means "not applicable").

Unpacker	Qihoo		Baidu	
	Partial Dex File	VM-protected DCode	Partial Dex File	VM-protected DCode
Android-unpacker [1]	✗	✗	✗	✗
drizzleDumper [7]	✓	✗	✗	✗
DexHunter [50]	✓	✗	✗	✗
PackerGrind [43]	✓	✗	✓	✗
`Parema`	✓	✓	✓	✓

Both `Android-unpacker` and `drizzleDumper` need to attach to the process of the target app through *ptrace()* but Baidu adopts anti-debugging technique to protect the packed apps from being attached through *ptrace()*. Thus, they cannot be applied to the apps packed by Baidu. For the apps packed by Qihoo, `Android-unpacker` outputs errors during unpacking and `drizzleDumper` dumps Dex files without VM-protected DCode.

`DexHunter` modifies the Android runtime and dumps Dex files using the system functions. Since Baidu hooks special system functions to protect the Dex data from being dumped, `DexHunter` cannot unpack apps packed by Baidu. For the apps packed by Qihoo, `DexHunter` only dumps the methods that are not VM-protected. `PackerGrind` also just recovers the DCode of methods that are not VM-protected in the apps packed by both packers.

Answer to RQ1: `Parema` can effectively locate PCode and PHs as well as learn the required information from training apps. Moreover, it can correctly recover both the DCode protected by traditional methods and VM-protected DCode of target apps, outperforming the off-the-shelf unpackers.

5.3 RQ2: Semantics Recovery

RQ2 evaluates whether `Parema` can recover the semantics of PHs in the VM-protected apps. If the semantics of PHs are correctly determined, we can recover the semantics executed by the P-VM according to the invoked PHs in the execution trace. To achieve this purpose, we first build the semantic features S_{ph} and S_{dh} by analyzing training apps and AOSPs, respectively, and then use them to recover the semantics of the PHs in the target VM-protected apps.

5.3.1 Semantic Features from PHs. After determining the PHs in §5.2.3, we run `Parema` to generate S_{ph} through symbolic analysis

Table 6: Semantics recovery result of four different VM-based packers (✓ represents "applicable" and ✗ means "not applicable").

	Qihoo		Baidu	
	VSetA	VSetB	VSetA	VSetB
Parema	✓	✓	✓	✓

Table 7: The cross validation of semantics recovery with the existing popular code similarity analysis tools (i.e., Genius [22], Gemini [42], Safe [28], and Parema). ●, ●, and ○ stand for the supports of complete, partial, and failed recovery, respectively.

Semantics	Genius	Gemini	Safe	Parema
move* except move-exception/result*	○	○	●	●
move-result* move-exception	○	○	○	●
const* except, const-method-* const-class/string*)	○	○	●	●
goto*	○	○	○	●
if-*	○	○	○	●
*switch	○	○	○	●
throw	○	○	○	●
return*	○	○	○	●
cmp-*	○	○	○	●
sh-int* *sh*-long*	○	○	○	●
neg-int, neg-long	○	○	●	●
not-int, not-long	○	○	●	●
-int, *-long* (except those on above)	○	○	○	●
-float *-double*	○	○	○	●
neg-float, neg-double	○	○	●	●
-to-	○	○	○	●
new-array	○	○	○	●
fill-array-data, aget*, aput*	○	○	○	●
invoke-*	○	○	○	●
iget*, iput*	○	○	○	●
sget*, sput*	○	○	○	●
monitor-*, etc.	○	○	○	●

using the approach described in §4.1. In learning phase, we just study training apps (i.e., TSetA) to obtain the required information. Then, for evaluation, we leverage the learnt S_{ph} to recover the semantics of the PHs in the target apps (i.e., VSetA and VSetB). More precisely, we first obtain the binary code of the PH through dynamic tracking (**U1** in Fig. 5) and build their semantic features (i.e., $\tilde{S}_{ph}$). Then, we recognize the semantics of all PHs according to the semantics similarities between S_{ph} and $\tilde{S}_{ph}$. We check the correctness of the recognized semantics by comparing the invoked PHs in the execution trace and the corresponding DCode of the original apps manually. The results listed in Table 6 show that the semantics of each PH can be correctly recovered.

5.3.2 Semantic Features from DHs. To evaluate the semantics recovery approach based on S_{dh}, we first build seven AOSPs (i.e., Android 4.4-10) and extract the DHs from their libraries libart.so (i.e., A-VMs). Then, we build S_{dh} using Parema through symbolic analysis of these DHs from all A-VMs collaboratively (approach in §4.2). All 22 supported types of semantic features (details in Table 2) are built. Instead of using S_{ph} in the evaluation of S_{ph}-based solution (§5.3.1), we leverage S_{dh} to recover the semantics of PHs, which are used for interpreting the PCode in the target apps. Specifically, we first build the semantic features of PHs (i.e., $\tilde{S}_{ph}$), and then recognize the semantics represented by $\tilde{S}_{ph}$ according to their semantic similarities to S_{dh} (approach details in §4.3.3). The results are shown in the right column of Table 7, where the symbol "●" indicates that Parema correctly recognizes the semantics of PHs in all apps packed by Baidu and Qihoo (i.e., VSetA and VSetB) statically and the symbol "●" stands for that the recovery of semantics also requires the information of the dynamically invoked runtime functions, which are already logged in the execution trace. Hence, for all these 22 types of semantics, Parema can recover them correctly.

For comparison, we also run three popular code similarity analysis tools (i.e., Genius [22], Gemini [42], and Safe [28]) to recognize the semantics of PHs based on the analysis of the code similarities between the implements of DHs and PHs. The results are also shown in Table 7, where the symbols "○" and "●" indicate that the corresponding tool can recognize the semantics of *no* and *partial* PHs, respectively. The comparison results show that Genius and Gemini cannot recognize the semantics of any PHs, and Safe just recognizes the semantics of several PHs. All these three tools aim to analyze the similarity between the binary code compiled from same source code with different configurations. Genius and Gemini focus on the analysis of the CFGs of binary code. Since there are many differences in the CFGs of PHs and DHs, both tools fail to identify the code similarities between PHs and DHs. Event though the CFGs of the PHs extracted from the AOSPs of different versions, they are different. Consequently, both Genius and Gemini fail to detect the similarities between the PHs of different AOSPs. Safe detects code similarities through first transforming the binary code into embeddings and then calculating the similarities of the embeddings. Since the embeddings of a few PHs and DHs are still similar, Safe detects the similarities successfully.

Answer to RQ2: Parema can effectively extract the semantic features of both PHs in training apps and DHs in AOSPs, and further leverage them to correctly detect the semantic similarities between different handlers for recovering the semantics of PHs (i.e., the PCode) in the target apps.

5.4 RQ3: Investigation of VM-based Packers

We investigate existing Android packers in three scenarios (i.e., D-1/2/3) using Parema. We first study the packers of seven publicly accessible Android packer providers, all of which claim to use VM-based protections, and find that only Baidu and Qihoo packers actually adopt VM-based protections. Hence, we focus on investigating the apps in VSetA and VSetB packed by two versions of Qihoo and Baidu packers.

D-1: Recovering Original DCode of VM-protected apps: In §5.3.1, in learning phase, we learn the symbolic expressions representing the decode/dispatch procedures (i.e., P2PH) of P-VMs from the training apps (i.e., TSetA) and the app-specific parameters f_{qihoo} and f_{baidu}. Then, during deobfuscation, we found their concrete values from the execution trace, and locate the PCode regions with help of the reverse-engineered PAM. Afterward, we calculate the address

Table 8: The number of the recovered instructions from the VM-protected *onCreate* methods of MainActivity. The first column shows the names of demonstrated apps and the second column represents he actual numbers of the DCode instructions. The left columns show the numbers of DCode instructions recovered from the apps packed by different versions (i.e., VSetA/VSetB) of the two VM-based packers (i.e., Qihoo and Baidu).

Packed apps	#Original instruction	Qihoo	Baidu
		VSetA/VSetB	VSetA/VSetB
com.uberspot.a2048	93	93/93	93/93
fr.asterope	150	150/150	150/150
net.mathdoku.holoken	373	373/373	373/373
net.tevp.postcode	29	29 /29	29/29
org.ligi.passandroid	109	109/109	109/109

of the PH for each PCode instruction by feeding the PCode into P2PH, construct the offset features/address (i.g., O_{ph}) of this PH, and further recover the VM-protected DCode with the learnt PH2D. By manually comparing the DCode recovered from the VM-protected target apps with their corresponding original DCode, we find that the VM-protected DCode of all target apps in VSetA are correctly recovered, whereas the VM-protected DCode of the apps in VSetB are not recovered because the packers packing the apps in VSetA are of the same version as those packing TSetA but the packers packing the apps of VSetB are of different versions.

For VSetA, we further reassemble the Dex files using the recovered DCode and reconstruct the app files. All these apps can be installed and run normally.

D-2: Semantics Recovery with Training Apps: We utilize the apps in VSetA and VSetB to study whether the semantics of the VM-protected DCode can be recovered with the knowledge (i.e., S_{ph}) learnt from training apps (TSetA). The evaluation results in Table 6 (§5.3.1) show that the executed semantics of the VM-protected code in the packed apps can be correctly recovered with S_{ph}. Table 8 shows the summarized recovery information of the VM-protected methods from five different VM-protected apps. As all these methods contain no branches, all their PCode instructions are executed during dynamic tracking. Meanwhile, *their semantics are correctly recovered.*

D-3: Semantics Recovery without Training Apps: For this investigation, we choose the knowledge (i.e., S_{dh}) learnt from AOSPs to recover the executed semantics of the packed apps in VSetA and VSetB, and the details are described in §5.3.2 (Table 7). From the results, we find *the semantics of the executed code can still be correctly recovered without training apps.*

Answer to RQ3: With training apps packed by the packers of the same version, all VM-protected DCode of the packed apps can be recovered (D-1); If the training apps are packed by the packers of different versions, the semantics of the executed code can still be recovered (D-2); Without any training apps, the semantics of the executed code can still be recovered with the knowledge learnt from the AOSPs (D-3). Overall, the investigating results demystify that existing VM-based packers do not provide adequate protection though they significantly raise the bar for unpacking,

5.5 Threat to Validity

External validity: One threat to the external validity is the requirement of training apps, which are needed by the techniques for D-1/D-2. However, almost all existing popular commercial packing services are publicly accessible, such as Baidu, Qihoo, Ijiami, and Bangcle. To mitigate such potential threat, we propose new semantics-based unpacking techniques for D-3, which learn the required semantic information from the DHs of AOSPs. Since AOSPs are open-sourced, these techniques do not need to access the VM-protected packers for building training apps, and they can be applied and generalized to investigating other VM-protected programs.

Another threat is although we studied 7 commercial packers, only two packers actually adopt VM-protections and the others just provide VM-based protection in their paid services, charging around USD 10,000 for packing each app. Due to the limited budget, we cannot conduct experiments on such paid packing services, but we believe Parema can be used to reveal their internals because they adopt similar VM-based techniques according to their introductions as well as our communications with the packing services providers.

Internal validity: The major threat to internal validity is the inherent code coverage limitation of dynamic analysis. Due to the heavy additional overhead introduced by interpreting PCode, only specific methods (e.g., *onCreate*) are VM-protected in the packed apps, and thus we mitigate such threat by triggering all these methods. We can also conduct static analysis on the recovered Dex file (e.g., looking for JNI invocations) to trigger the execution of unexplored code interactively until all VM-protected code are recovered. Xue et al. [43] showed that such an adaptive unpacking procedure is useful to handle the traditional Android packers.

6 LIMITATION AND SUGGESTION

Unpacking apps protected by arbitrary VM-based protection without any information is still an open problem. As the first study on unpacking VM-protected apps, we discuss the limitations of our solution and possible solutions as follows. We also suggest methods to enhance the existing VM-based packers, which shed light on the future research on this topic.

First, we assume that one type of DCode instruction is translated into a fixed number of PCode instructions, and each PCode instruction is interpreted by a register-based interpreter, but the advanced packers can adopt a much more complex translation policy and implement a sophisticated interpreter. However, the packing services do not change the semantics of the code and thus all PCode in the VM-protected apps are semantically equivalent to original DCode. Therefore our semantics-based investigation techniques could still recover the semantics of the VM-protected code. Moreover, we can enhance our techniques to reverse-engineer the P-VM as long as we can get the VM-protected apps.

Second, although Parema has automated the majority of steps for unpacking the VM-protected Android apps, some manual efforts are still required, such as determining the actual decode/dispatch procedures and identifying the syntax of the PCode instructions. However, such analysis just needs to be conducted once for one packer or instruction. Also, in future work, we will explore machine learning based approaches to fully automate the analysis.

Third, we assume that the packer is known during investigation. This assumption is rational because the packers usually have obvious fingerprints in their packed apps, e.g., embedded libraries, methods, classes, and shell code, etc. We can also use the packer recognition approaches proposed in [44] to identify the packers. Also, the fingerprints of the same packer do not change much in different versions. According to our analysis results, the packers of two different versions usually have similar VM-protection technologies and implementation patterns but different code translation rules. In this paper, we do not focus on packer recognition. In future work, we will explore machine learning based approaches to recognize the packers with high robustness.

Enhancing VM-based Protection: By analyzing the latest publically available VM-based Android packers, we find that although they increase the bar of unpacking, the packed apps can still be unpacked potentially. These VM-based Android packers adopt one-to-one mapping between the DCode instructions and the PCode instructions, and the PCode instructions adopt similar syntax as the DCode instructions. Although they add app-specific parameters to the decoding and dispatching processes, the factors can be first recognized by comparing the decoding and dispatching processes of the different training apps and then identified from the execution trace of the target apps during recovering. Other techniques can be used by these packers to enhance their protection capability. For example, they can translate one DCode instruction into diverse PCode instructions that have completely different syntaxes from the PCodee instruction. They can also add the instruction-specific factors instead of the app-specific parameters. Moreover, applying the app-specific `P-VM` to the packed apps can also make the packers more sophisticated.

7 RELATED WORK

Android (Un)Packing Techniques: Various Android packers have been developed to protect apps from being analyzed and repackaged [6, 14–16, 26, 29, 32, 38, 48]. The majority of them follow the write-and-then-execute rule. That is, they will encrypt/hide the Dex data statically and then dynamically release the Dex data into the memory during the execution. Hence, the protected Dex data cannot be found by reverse-engineering the apps statically. Exploiting this observation, existing unpacking approaches [21, 30, 40, 43, 47, 50] look for and dump the Dex data in the memory. However, they cannot unpack VM-protected apps because the original Dalvik bytecode is never released into the memory.

PC (Un)packers: The PC programs are implemented in native instructions (e.g., x86/ARM instructions) [18, 24, 34, 39]. They are translated into bytecode during VM-based packing. In contrast, Android apps are implemented in Dalvik bytecode, which is translated into customized types of bytecode (PCode) during VM-based packing. Although semantic information is considered in [35], it just refers to the control flow of the bytecode in packed native programs. In this paper, the semantics mainly represents the functionalities of the original Dalvik bytecode (i.e., DCode). Since Android has a multiple-layer architecture (e.g., Linux kernel, HAL, runtime, framework, and apps) [45], the cross-layer behaviors (e.g., JNI invocations) are commonly used by Android packers for protection, but the packers for desktop programs have no such behaviors.

Unpacking VM-Protected Binaries: VM-protection technique was first employed to protect desktop programs [3–5, 23]. Several approaches have been proposed to facilitate the analysis of VM-protected binaries [17, 19, 27, 33, 35, 46]. They can be generally divided into three categories. First, some approaches aim at reverse-engineering the VM, either manually [33] or automatically [35]. Second, some methods target on reconstructing the CFGs of the original programs by simplifying away obfuscation code through equational reasoning or semantics-preserving program transformations[19, 46]. Third, some systems try to generate the easy-to-read pseudo-code instructions of the translated PCode. For example, the VMAttack [27] maps complex bytecode sequences of the VM to easy-to-read pseudo-code instructions. Blazytko *et al.* [17] leverages program synthesis techniques to generate code that approximates the semantics of the original program protected by VM. However, these approaches cannot handle complex, non-linear expressions, let alone complicated Android apps.

Code Similarity Detection: Code similarity detection approaches are already widely used in bug detection, malicious code identification, and so on [22, 25, 28, 42, 49]. However, since these approaches are designed with the purpose of detecting code similarity between the same or similar binary code, they are not suitable for semantics similarity detection between the various binaries implemented by different developers. In this paper, to investigate the VM-based Android packers involving various types of code, we propose `Parema` supporting semantic similarity analysis of different types of code.

8 CONCLUSION

In this paper, we took an important step to catch up with packers because VM-based packing techniques render existing unpackers ineffective. To this end, we propose a novel investigation approach to deobfuscate the VM-protected DCode of the apps packed by VM-based packers with knowledge learnt from training apps or AOSPs under three different scenarios, including the original DCode recovery with training apps packed by the packers of the same version to that packing the target apps, the original semantics recovery with the training apps packed the packers of different versions from that packing the target apps, and the original semantics recovery without training apps. Moreover, to assist the investigation, we develop a prototype named `Parema` after tackling a number of challenging issues. The evaluation results show that though the existing VM-based packers provide stronger protection than traditional packers, the DCode protected by VM and/or traditional methods in their packed apps can still be unpacked and deobfuscated by `Parema`.

ACKNOWLEDGMENT

We sincerely thank Dr. Chennian Sun for shepherding our paper and the anonymous reviewers for their constructive comments. We thank Prof. Zhiqiang Lin for his assistance during preparing this paper. This work is partly supported by Hong Kong RGC Projects (No. 152223/17E), NSFC Young Scientists Fund (No. 62002306), HKPolyU Start-up Fund (ZVU7), CCF-Tencent Open Research Fund (ZDCK), 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] 2014. Android-unpacker. https://github.com/strazzere/android-unpacker.
[2] 2015. F-Droid. https://f-droid.org/.
[3] 2016. ASProtect. http://www.aspack.com/asprotect32.html.
[4] 2016. Code virtualizer. http://www.oreans.com/codevirtualizer.php.
[5] 2016. VMProtect. http://vmpsoft.com.
[6] 2017. APK Protect. https://sourceforge.net/projects/apkprotect.
[7] 2017. drizzleDumper. https://github.com/DrizzleRisk/drizzleDumper.
[8] 2018. JNI Tips. https://developer.android.com/training/articles/perf-jni.
[9] 2018. Valgrind. http://valgrind.org/.
[10] 2019. angr. https://github.com/angr/angr.
[11] 2020. The Data Sets. https://www.dropbox.com/sh/sf59dtfsfthlv5i/AAC0wQXo-J94y_0TiTqz7un0a?dl=0.
[12] 2020. Z3. https://pypi.org/project/z3-solver/.
[13] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.
[14] Alibaba Inc. 2017. http://jaq.alibaba.com/.
[15] Baidu Inc. 2017. http://app.baidu.com.
[16] Bangcle Inc. 2017. http://www.bangcle.com/.
[17] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the semantics of obfuscated code. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. 643–659.
[18] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 745–756.
[19] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the ACM conference on Computer and communications security (CCS)*. 275–284.
[20] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 37–54.
[21] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[22] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 480–491.
[23] Sudeep Ghosh, Jason D Hiser, and Jack W Davidson. 2010. A secure and robust approach to software tamper resistance. In *Proceedings of the International Workshop on Information Hiding (IH)*. Springer, 33–47.
[24] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. 2008. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 98–115.
[25] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 57–67.
[26] Ijiami Inc. 2017. http://www.ijiami.cn.
[27] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: deobfuscating virtualization-based packed binaries. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. 1–10.
[28] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 309–329.
[29] NAGA IN Inc. 2017. http://www.nagain.com/.
[30] Zhenyu Ning and Fengwei Zhang. 2018. DexLego: Reassembleable bytecode extraction for aiding static analysis. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 690–701.
[31] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists (IMECS)*, Vol. 1. 380–384.
[32] Qihoo360 Inc. 2017. http://dev.360.cn/.
[33] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *Proceedings of the USENIX Workshop on Offensive Technologies.(WOOT)*.
[34] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–37.
[35] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE, 94–109.
[36] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.
[37] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc.
[38] Tencent Inc. 2017. https://www.qcloud.com/product/cr.
[39] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[40] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 1247–1262.
[41] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 442–458.
[42] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 363–376.
[43] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 358–369.
[44] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Transactions on Software Engineering* (2020).
[45] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for {ART}. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 289–306.
[46] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 674–691.
[47] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 359–381.
[48] Rowland Yu. 2014. Android packers: facing the challenges, building solutions. In *Proceedings of the 24th Virus Bulletin International Conference (VB)*.
[49] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 34. 1145–1152.
[50] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: toward extracting hidden code from packed android applications. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*. Springer, 293–311.
[51] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. 2014. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 199–210.
[52] Yajin Zhou. 2017. The evolution of Android app packing and unpacking techniques. Hitcon.