REFERENCES

[1] 2014. Android-unpacker. https://github.com/strazzere/android-unpacker. (2014).
[2] 2015. F-Droid. https://f-droid.org/. (2015).
[3] 2016. ASProtect. http://www.aspack.com/asprotect32.html. (2016).
[4] 2016. CF-Bench. http://http://bench.chainfire.eu/. (2016).
[5] 2016. Code virtualizer. http://www.oreans.com/codevirtualizer.php. (2016).
[6] 2016. Dalvik Executable format. https://source.android.com/devices/tech/dalvik/dex-format.html. (2016).
[7] 2016. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html. (2016).
[8] 2016. VMProtect. http://vmpsoft.com. (2016).
[9] 2017. APK Protect. https://sourceforge.net/projects/apkprotect. (2017).
[10] 2017. CLOC: Count Lines of Code. http://cloc.sourceforge.net/. (2017).
[11] 2017. drizzleDumper. https://github.com/DrizzleRisk/drizzleDumper. (2017).
[12] 2018. JNI Tips. https://developer.android.com/training/articles/perf-jni. (2018).
[13] 2018. Valgrind. http://valgrind.org/. (2018).
[14] 2019. The Supplemental Material for Paper #169. (2019).
[15] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.
[16] Alibaba Inc. 2017. http://jaq.alibaba.com/. (2017).
[17] Kathy Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proc. CCS*.
[18] Baidu Inc. 2017. http://app.baidu.com. (2017).
[19] Bangcle Inc. 2017. http://www.bangcle.com/. (2017).
[20] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. SOAP*.
[21] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Proc. Security*.
[22] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proc. CCS*.
[23] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets.. In *Proc. ESORICS*.
[24] 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 *Proc. NDSS*.
[25] Sudeep Ghosh, Jason D Hiser, and Jack W Davidson. 2010. A secure and robust approach to software tamper resistance. In *Proc. IH*.
[26] Ijiami Inc. 2017. http://www.ijiami.cn. (2017).
[27] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proc. ARES*.
[28] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proc. FSE*.
[29] NAGA IN Inc. 2017. http://www.nagain.com/. (2017).
[30] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. ACM PLDI*.
[31] Qihoo360 Inc. 2017. http://dev.360.cn/. (2017).
[32] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *Proc. USENIX WOOT*.
[33] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Proc. IEEE SP*.
[34] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. 2008. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)* 4, 4 (2008).
[35] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.
[36] 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.
[37] Tencent Inc. 2017. https://www.qcloud.com/product/cr. (2017).
[38] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. In *Proc. ACM CCS*.
[39] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive Unpacking of Android Apps. In *Proc. ICSE*.
[40] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *Proc. IEEE SP*.
[41] L. Yan and H. Yin. 2012. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. USENIX Security*.
[42] 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 *Proc. RAID*.
[43] Rowland Yu. 2014. Android packers: facing the challenges, building solutions. In *Proc. VB*.
[44] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. DexHunter: toward extracting hidden code from packed Android applications. In *Proc. ESORICS*.
[45] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. 2014. DIVILAR: Diversifying Intermediate Language for Anti-repackaging on Android Platform. In *Proc. ACM CODASPY*.
[46] Yajin Zhou. 2017. The evolution of Android app packing and unpacking techniques. Hitcon. (2017).

Supplemental Material for Submission #169

Anonymous Author(s)

ABSTRACT

Android packers have been widely adopted by developers to protect apps from being plagiarized. At the same time, attackers have abused packers to "protect" malware and evade the detection. Although recent proposed methods could *unpack* apps protected by traditional packers through direct memory dumping, the quickly-evolving commercial packers adopt a virtual machine (VM) based protection, which stores the customized bytecode instead of the original Dalvik bytecode in the memory. This defeats memory dumping mechanism. However, little is known about whether such packers can provide enough protection to benign apps, or whether we can still analyze the malware "protected" by them.

In this paper, we aim to shed light on these questions, and take the first step towards recovering the original Dalvik bytecode from VM-protected apps and reconstructing the Dex file, which can be further analyzed by the off-the-shelf tools. In particular, we propose a novel semantics-based solution consisting of two phases: a training phase to collect necessary information of a target packer's VM and a recovering phase to recover the original Dalvik bytecode by leveraging the obtained information in the training phase. We develop a prototype named Parema and evaluate it through commercial VM-based Android packers and packed malware. The experimental result demonstrated its effectiveness.

A MOTIVATING EXAMPLE

To illustrate the basic idea of VM-based packing technique in this section and that of our semantics-based unpacking approach in §B), we design an example following the basic packing mechanism of the VM-based packer in [45].

Fig. 8 shows the method *vmpFunc()* before and after being protected. This function first assigns constants 20 and 25 to the integer variables `a` and `b`, respectively, and then assigns the summarization of these two variables to the variable `s`. As shown in Fig. 4, the packing process consists of two steps: (1) turning the DCode to the intermediate bytecode. In particular, the opcodes are changed according to the fixed rules whereas the operands are unaltered. For example, the opcode `0x13` is converted to `0xca`; (2) converting the intermediate bytecode to PCode with an app-specific parameter through `xor`. Since the second step introduces the app-specific parameter, there is no fixed mapping relationship between the PCode instructions and the DCode instructions for all apps protected by this packer.

Moreover, as shown in Fig. 8, the original method *vmpFunc()* is replaced by a native method that severs as the entry of the context switch between PVM and Android runtime. That is, the execution context goes into PVM for interpreting the PCode instructions of the VM-protected method when the method is invoked, and returns to Android runtime from PVM when the method returns.

We also implement an indirect-threaded interpreter (i.e., Fig. 2(b)) for this VM-based packer, and show two handlers (i.e.,*PHandler_Const16()* and *PHandler_AddInt()*) in Fig. 9, which are

Original implementation	After VM-based obfuscation

`void vmpFunc(...) {`
`  int a = 20, b = 25;`
`  int s = a + b;`
`  ...`
`}`
	`native void vmpFunc(int m_id);`

Figure 8: The method before and after VM-protection in Dex file.

used to interpret the PCode instructions in method *vmpFunc()*. Being invoked by *vmpFunc()*, the function *execute()* (Line 43) is the entrance of the interpreter and its argument `m_id` represents ID of the invoked VM-protected method, which is used by the interpreter to locate the PCode of the method to be interpreted. More precisely, the interpreter first locates the memory region `PCode` storing the PCode of the target method by invoking function *lookup_pcode()* (Line 47) and then obtains the app-specific parameter `pwd` through function *init_decoding_factor()* (e.g., `0x8080`) (Line 49). The virtual program register `vpc` points to the PCode to be decoded.To interpret the PCode instruction indexed by `vpc` (Line 14, 34, and 52), the interpreter firsts recover the intermediate bytecode by calculating the `xor` of PCode and the app-specific parameter (i.e., `pwd`), and then calls the proper PHandlers according to the opcodes (Line 16, 35, 49 in Fig. 9) of the intermediate bytecode.

This interpreter utilizes an array to emulate the virtual registers (i.e., Line 11 and 31in Fig. 9). Since this interpreter is indirect-threaded for better performance than the decode-dispatch interpreter, each PHandler has its own decode-dispatch procedure, such as Line 5-14 in *PHandler_Const16()* and Line 24-34 in *PHandler_AddInt()*. Hence, there is no a decode-dispatch loop in the CFG, which has been used to reverse engineer PVM for protecting desktop programs [33].

To explain how PCode is executed by PVM as shown in Fig. 9, We take the first instruction `c208 081c` (e.g., `const/16 v0, #int 20`) as an example and list the corresponding results in Fig. 10. The first bytecode `c208` is translated to the intermediate code `ca00` with the app-specific parameter `pwd` (i.e., `0x0808`) (Line 52). Then, according to the recovered opcode `ca`, the corresponding PHandler *PHandler_Const16* is called (Line 55). In *PHandler_Const16*, after `xor`ing the two bytecodes and the app-specific parameter (Line 5-6 in Fig. 9), we obtain the operands `0x00` and `0x0014` (Line 8-9) according to the syntax of PCode instruction `Const/16`, which is represented by opcode `ca`. Finally. the semantics of this instruction is executed by the PHandler with the operands (Line 11).

B EXAMPLE

We use the motivating example in §A to illustrate our solution of recovering the VM-protected DCode.

Training.: To reverse engineer the interpreter shown in Fig. 9 and learn the rules (Table 1) for recovering the VM-protected DCode, we first implement several training apps that put all possible DCode instructions into the methods to be protected by the VM-based packer and then upload them to the packer's server. After obtaining the VM-protected training apps, we run them and collect the

```
1 int vRegs[]; // The virtual register array
2 int PHandlers[]; // The PHandler table
3 int pwd;     // The app-specific decoding factor
4 void PHandler_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  vreg  = bytecode1 & 0xff;
9   int16_t value = bytecode2;
10  // Execute the semantics
11  vRegs[vreg] = value;
12  vpc += 2; // Point the next instruction
13  // Decode  the next instruction
14  int8_t next_opcode = (Code[vpc] ^ pwd & 0xff00) >> 0x8;
15  // Dispatch PHandler for the next instruction
16  void (*pHandler)(int, int16_t*);
17  pHandler = (void(*)(int, int16_t*))PHandlers[next_opcode];
18  if (pHandler != NULL):
19    pHandler(vpc, PCode); // Invoke the target PHandler
20  return;
21 }
22
23 void PHandler_AddInt(int vpc, int16_t PCode[]) {
24  int16_t bytecode1 = PCode[vpc] ^ pwd;
25  int16_t bytecode2 = PCode[vpc+1] ^ pwd;
26  // Parse the operands
27  int8_t  vreg1 = bytecode1 & 0xff;
28  int8_t  vreg2 = (bytecode2 & 0xff00) >> 0x8;
29  int8_t  vreg3 = bytecode2 & 0xff;
30  // Execute the semantics
31  vRegs[vreg1] = vRegs[vreg2] + vRegs[vreg3];
32  vpc += 2; // Point to the next instruction
33  // Decode the next instruction
34  int8_t next_opcode = (PCode[vpc] ^ pwd & 0xff00) >> 0x8;
35  // Dispatch PHandler for the next instruction
36  void (*pHandler)(int, int16_t*);
37  pHandler = (void(*)(int, int16_t*))PHandlers[next_opcode];
38  if (pHandler != NULL):
39    pHandler(vpc, PCode); // Invoke the target PHhandler
40  return;
41 }
42 // Argument is the ID of the target vm-protected method
43 void execute(int16_t m_id) {
44  // Initialize the virtual PC, registers and the handler table
45  int vpc = 0;
46  // Lookup the PCode of the target method according to m_id
47  int16_t *PCode = lookup_pcode(m_id);
48  init_virtual_registers(vRegs);
49  init_decoding_factor(&pwd); // pwd=0x0808 in this example
50  init_PCode_handlers(PHandlers);
51  // Decode the first instruction
52  int8_t opcode = (PCode[vpc] ^ pwd & 0xff00) >> 0x8;
53  // Dispatch PHandler for the first instruction
54  void (*pHandler)(int, int16_t*);
55  pHandler = (void(*)(int, int16_t*))PHandlers[opcode];
56  if (pHandler != NULL):
57    pHandler(vpc, PCode); // Invoke the target PHandler
58  return;
59 }
```

Figure 9: An indirect-threaded interpreter provided by the PVM to interpret the PCode instructions, and the implementations two handlers (i.e., *PHandler_Const16()* and *PHandler_AddInt()*) are shown.

	PCode	App-specific factor	Intermediate code	PHandler	Operands
1	c208 081c	^ 0x0808	ca00 0014	PHandler_Const16	0x00, 0x0014
2	c209 0811		ca01 0019	PHandler_Const16	0x01, 0x0019
3	8b0a 0809		8302 0001	Phandler_AddInt	0x02, 0x00, 0x01
	...		...	...	...

Figure 10: The decode-dispatch procedure of the VM-protected methods when the packed app runs on the smartphone.

execution trace. Then, we identify the rules (i.e., `PH2D`, `P2PH` and `PAM`) by analyzing the execution traces through step ①-⑤ in Fig. 3.

`PAM` represents the PCode addressing mechanism implemented in the function *lookup_pcode()* (Line 47 in Fig. 9), and `P2PH` refers to the decode-dispatch procedure (Line 14, 34 and 52 in Fig. 9). `PH2D`, the mapping from PHandler to the DCode instruction, is learnt through comparing the DCode instructions in the original training apps and the PHandlers of the corresponding PCode instructions in the packed versions. For example, the "`const/16`" DCode instruction is translated to a "`const/16`" PCode instruction in the packed app, which will be executed by the function *PHandler_Const16()*. Hence, if *PHandler_Const16()* is invoked, it means the original DCode instruction is "`const/16`". Moreover, by analyzing the execution trace of *PHandler_Const16()*, we can further revere engineer the syntax of PCode, of which the low byte of the first bytecode and the second bytecode are the operands (Line 8-9 in Fig. 9).

Recovering.: To recover the VM-protected DCode of a target app packed by this packer, `Parema` first runs to collect the required information, such as the `pwd` (Line 49 in Fig. 9) and `PHandlers` (Line 50 in Fig. 9) involved in the decode and dispatch procedure. In step ❶, `Parema` leverages `PAM` to find the PCode of this VM-protected method according to the method ID (i.e., `m_id`). In step ❷, `Parema` utilizes `P2PH` to generate the offset signatures of the PHandlers with the PCode, `pwd` and `PHandlers`. In step ❸, `Parema` recovers the DCode according to offset signatures of the PHandlers based on `PH2D`.

C IMPLEMENTATION

We implement our solution in a prototype named `Parema`, which has a dynamic tracking module and a static analysis module, with around 9k lines of C/C++ code and 3k lines of Python script, calculated by CLOC [10]. `Parema` adopts the similar mechanism of PackerGrind [39] to dynamically collect the DCode in the memory, which is not protected by the VM-based approach. Specifically, we wrap the function ART runtime functions (i.e., *ArtMethod:Invoke()*) and instrument the memory modification statements (i.e., `IR_Load` and `IR_Store`) using Valgrind. Then we collect the Dex items in the wrapping function by parsing the Dex files in memory [6]. We reconstruct the original Dex file by combining the DCode recovered from VM-protected methods and that collected from the memory.

C.1 Dynamic Tracking Module

Based on `Valgrind` [30], the dynamic tracking module monitors the behaviors of a packed app, logs its execution trace, and collects Dex data without VM-based protection. `Parema` runs the packed apps on real mobile devices, and collects the execution trace. To track the IR statements and expressions, we insert IR statements before each IR statement to invoke a helper function for instrumentation. `Parema` logs all the executed IR statements and the following information from Android framework, Android runtime, and the underlying Linux system to facilitate the analysis and DCode recovery: (1) The events in Android framework (e.g., invocations of the Java methods in an app and/or the Android framework). Such information helps us locate and verify the PHandlers; (2) The events in Android runtime and the invocations of JNI reflection functions. For instance, by tracking the invocation of *art_quick_generic_jni_trampoline*, we know that the execution context switches from Android runtime to PVM. Moreover, the JNI reflection functions help us identify and analyze the method invocation instructions; (3) The events in the underlying Linux system, including (a) the memory related operations, such as *malloc()*, *free()*, *sys_map()*, and *sys_unmap()*), which

Table 8: The performance evaluation with CF-Bench [4].

	Native Scores			Java Scores			Overall Scores		
	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown
Base	29285.53	1825.26		10570.57	2084.82		18056.20	1690.00	
Valgrind	2740.33	167.49	10x	2416.00	145.24	4x	2545.33	151.45	7x
Parema	753.67	43.37	39x	674.79	36.27	15x	709.97	40.71	25x

will be used for generating the offset signatures of the PHandlers. (b) The modifications and translations of the data in the memory. These information can be obtained by monitoring the data translation functions (e.g., *strcpy()* and *memcpy()*), and they will help us locate the interested data, such as the Dex data items. (c) Other special behaviors of the packers, such as *sys_ptrace()* for anti-debug and *sys_exec()* to execute special commands. We collect them by monitoring the system calls.

C.2 Static Analysis Module

The static analysis module aims at accomplishing two major tasks. First, in the training phase, it follows the approach introduced in §**??** to recognize `P2PH` and `PH2D` and identify `PAM` by analyzing the execution trace of the VM-protected training apps. Second, in the recovering phase, it first recovers the original DCode of the target apps following the mechanism in §3.3.1 and §3.3.2, and then utilizes the mechanism in §3.3.3 to reconstruct the Dex files with the recovered DCode and the Dex data collected from the memory.

D PERFORMANCE

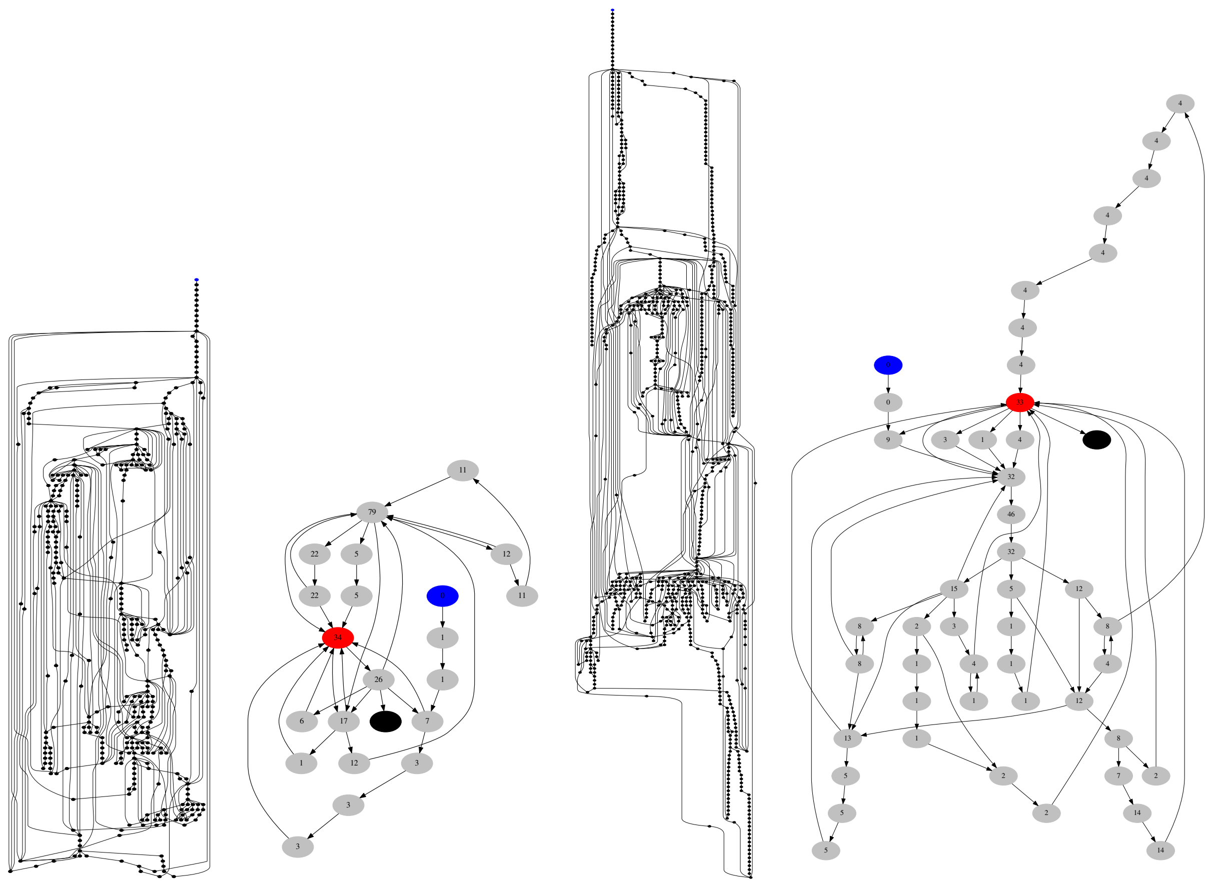
We evaluate the overhead of `Parema`'s dynamic tracking subsystem through running CF-Bench [4] without DBI (Base), with only Valgrind, and with `Parema`, respectively, From the results shown in Table 8, we observe that `Parema` brings 39x and 15x slowdown to the native score and Java score receptively, and the overall slowdown is 25x. Note that even we just run Valgrind without `Parema`, it introduces 7x overall slowdown. Therefore, `Parema` just brings 3.7 (25x/7x) additional overall slowdown to Valgrind. Compared with the emulator-based DBI systems [41] that bring 11-34x slowdown, the performance of `Parema` is reasonable and acceptable.

E ENHANCING VM-BASED PROTECTION

By analyzing the latest publically available VM-based Android packers (i.e., Qihoo and Baidu), we find, although they obviously increase the bar of unpacking, the apps packed by these packers are still possible to be unpacked by `Parema`. 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 decode-dispatch processes, the factors can also be recognized by comparing the decode-dispatch processes of the different training apps and then identified from the execution trace of the target apps during recovering. There are still various techniques that the packers can utilize to become more secure. For example, they can translate one DCode instruction into various PCode instructions, which have completely different syntaxes from the PCode instruction. They can also add the instruction-specific factors instead of the app-specific parameters. In addition, applying the app-specific PVM to the packed apps can also make the packers more sophisticated.

F CFGS VS. SEGS

Fig. 11 shows the CFGs and the SEGs generated from the execution trace of a training app packed by Qihoo and Baidu, respectively. The two CFGs contain 396 nodes/509 edges (Fig. 11(a)) and 835 nodes/979 edges (Fig. 11(c)), separately. By contrast, there are only 48 nodes/69 edges and 22 nodes/35 edges in the SEGs as shown in Fig. 11(b) and Fig. 11(d).



(a) CFG of Qihoo packer (b) SEG of Qihoo packer (c) CFG of Baidu packer (d) SEG of Baidu packer

Figure 11: The generated CFG and SEG of two different packers, of which Qihoo adopts *decode-dispatch* interpreter and Baidu adopts *indirect threaded* interpreter. The red node in the SEG is the identified entry nodes of the decode-dispatch procedure. The number inside the node is the in-degree of the node.