VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification

Dongpeng Xu
Pennsylvania State University
University Park, PA 16802, USA
dux103@ist.psu.edu

Jiang Ming
University of Texas at Arlington
Arlington, TX 76019, USA
jiang.ming@uta.edu

Yu Fu
Pennsylvania State University
University Park, PA 16802, USA
yuf123@ist.psu.edu

Dinghao Wu
Pennsylvania State University
University Park, PA 16802, USA
dwu@ist.psu.edu

ABSTRACT

Code virtualization is a highly sophisticated obfuscation technique adopted by malware authors to stay under the radar. However, the increasing complexity of code virtualization also becomes a "double-edged sword" for practical application. Due to its performance limitations and compatibility problems, code virtualization is seldom used on an entire program. Rather, it is mainly used only to safeguard the key parts of code such as security checks and encryption keys. Many techniques have been proposed to reverse engineer the virtualized code, but they share some common limitations. They assume the scope of virtualized code is known in advance and mainly focus on the classic structure of code emulator. Also, few work verifies the correctness of their deobfuscation results.

In this paper, with fewer assumptions on the type and scope of code virtualization, we present a verifiable method to address the challenge of partially-virtualized binary code simplification. Our key insight is that code virtualization is a kind of process-level virtual machine (VM), and the context switch patterns when entering and exiting the VM can be used to detect the VM boundaries. Based on the scope of VM boundary, we simplify the virtualized code. We first ignore all the instructions in a given virtualized snippet that do not affect the final result of that snippet. To better revert the data obfuscation effect that encodes a variable through bitwise operations, we then run a new symbolic execution called *multiple granularity symbolic execution* to further simplify the trace snippet. The generated concise symbolic formulas facilitate the correctness testing of our simplification results. We have implemented our idea as an open source tool, *VMHunt*, and evaluated it with real-world applications and malware. The encouraging experimental results demonstrate that VMHunt is a significant improvement over the state of the art.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3243827

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

Code Virtualization, Binary Analysis, De-obfuscation, Multiple Granularity Symbolic Execution

ACM Reference Format:
Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243827

1 INTRODUCTION

Virtualization, a general technique that runs a virtual machine on versatile platforms [63], has become an important technique for software protection to obfuscate code [5, 23, 62]. When applied to code obfuscation, virtualization transforms the selected parts of a program to bytecode in a new, custom virtual instruction set architecture (ISA). At execution time, the bytecode is emulated by an embedded virtual machine (or interpreter) on the real machine. The new ISA can be designed independently, and thus the bytecode and interpreter greatly differ from those in every protected instance. In this way, the program's original code never reappears. Moreover, the bytecode is typically implemented in a RISC-like style, in which a source x86 instruction will be translated to a sequence of bytecode operations. Consequently, the number of native instructions executed increases significantly [36], and extracting the semantics of the custom ISA is like finding a needle in a haystack. Furthermore, virtualization obfuscation can be seamlessly integrated with other obfuscation schemes such as data encoding [16, 86], metamorphism [1, 85], and control flow obfuscation [17, 74], rendering traditional static and dynamic analysis techniques ineffective [13, 32]. Over the last decade virtualization obfuscation, generally recognized as one of the most advanced techniques to impede reverse engineering [39, 50], has been developed as a set of commercial software protection products [45, 46, 66, 69, 73] and research tools [15, 53, 68].

The potency of virtualization obfuscation has definitely attracted the attention of malware developers, who are highly motivated

to seek more sophisticated techniques for disguising their malicious code and circumventing anti-virus solutions [43, 54]. An increasing number of malware are armored by virtualization such as Virus.Win32.Goblin [47] and Trojan.Win32.Clampi [26]. However, the heavy-weight obfuscation strength also comes with a cost of performance and compatibility. Because the virtualized code does not execute natively, its runtime overhead is considerably high. A recent study shows that the slowdown varies from 1.9X to 660.9X when only 10% of the code is virtualized [77]. The user manual of Code Virtualizer [44] recommends that users only protect a short, core code snippet and avoid program hot spots such as loop structures that iterate many times. Another limitation is the compatibility issue. Some complicated program structures (e.g., switch/case statements and exception handlers) and x86 instructions (e.g., SIMD and AES instruction set) might not be correctly translated by the virtualization tools [55]. The same study [77] also confirms the compatibility problem in that most test programs exit exceptionally when virtualization obfuscation level reaches 30%. Therefore, unlike the binary packers, another common obfuscation scheme for whole binary code protection [56, 71], malware developers only virtualize selected, key parts of the malicious code in practice [78].

Most of existing work relies on dynamic analysis to deobfuscate virtualized program. The first category of work attempts to reverse engineer the bytecode interpreter [29, 30, 51, 55, 59]. Dynamic analysis is used to identify the decode-dispatch based interpretation, which is the classic way to implement an instruction set virtualization [63]. The distinguishing feature is a central loop that fetches a piece of bytecode based on the current value of a virtual program counter (i.e., decode), and then dispatches to the corresponding handler which contains the machine code to emulate the bytecode. Since bytecode handlers themselves are usually heavily obfuscated, they need to be further simplified, e.g., by program synthesis [9], symbolic execution [10, 37], or compiler optimizations [22, 57, 80]. The second category tries to strip off the virtualization obfuscation layer from the tedious execution instructions [19, 40, 67, 83]. In this category, dynamic taint analysis or concolic execution is applied to identify the instructions that contribute to the real program behavior. However, a common limitation in both categories is that they assume the scope of virtualization-obfuscated code is already known. None of them discusses how to automatically extract the virtualized code part from an obfuscated program. Besides, the way of designing ground truth dataset in prior evaluations is biased either: they perform whole program virtualization on very tiny, synthetic programs [9, 19, 30, 59, 83]. Unfortunately, this assumption may not be tenable for real-world applications where only part of the code is virtualized.

Automatic detection of the virtualized code is an indispensable step before deobfuscation procedures. Accurate boundaries can quickly locate virtualized code and significantly reduce the overhead, as deobfuscation procedures are generally quite expensive. However, locating accurate boundaries is a nontrivial task. First, decode-dispatch loops appear in many normal applications such as web servers and user interfaces. Second, advanced code emulators have adopted alternative interpretation structures, such as threaded interpretation [8, 24], to hide the decode-dispatch features. In threaded interpretation, the dispatch loop is inlined to the individual handler functions. Worse still, we have observed fake decode-dispatch loops that look very similar to real ones. Those loops usually contain an increasing integer variable, which mimics the virtual program counter inside the dispatcher. Their purpose is to mislead the deobfuscation methods that rely on looking for the decode-dispatch loop [29, 30, 37, 51, 55, 59].

In this paper, we propose a novel method, called *VMHunt*, to automatically identify and simplify virtualized code sections from an execution trace. VMHunt does not assume any particular structure of the bytecode interpreter in use. Instead, it locates the boundary of partially-virtualized code based on an inherent property of standard virtual machine (VM) design: *context switches occur between virtualization application and native OS to ensure isolation [33].* The code snippet inside a virtualized context is the virtualized code. With the boundary information, VMHunt extracts the core part of virtualized code by slicing the instructions that affect the native context. Finally, VMHunt includes a new method called *multiple granularity symbolic execution* to further simplify the sliced code. Compared to the traditional symbolic execution, our design can better revert data encoding effect and lead to more concise symbolic formulas, which represent the original semantics of the virtualized code. Furthermore, with the unprotected version of code, we can use the generated symbolic formulas to verify the correctness of VMHunt. Any semantic discrepancy may indicate that VMHunt is not implemented perfectly right.

We have evaluated VMHunt on the latest version of well-known virtualization obfuscators such as Code Virtualizer [45], VMProtect [73], EXECryptor [66], and Themida [46] in benign and malicious scenarios. Our experiment shows that VMHunt correctly extracts the virtualized section from a tedious execution trace without false positives. VMHunt's trace simplification can reduce the number of inflated instructions by several orders of magnitude, and multiple granularity symbolic execution delivers a significantly concise form which accurately reveals the semantics of the virtualized code. Our experiments demonstrate VMHunt enables an accurate analysis and understanding of virtualization-obfuscated binary code, and it is essential for rapid response to emerging malware threats.

Scope and Contributions. VMHunt does not attempt to supersede existing virtualization deobfuscation methods, but rather complements them by narrowing down the search scope for virtualized code snippets, and then providing a simplified view of those snippets. Other state-of-the-art techniques such as program synthesis [9] and compiler optimizations [57] can work directly on VMHunt's simplified trace and achieve better reverse engineering result. In summary, we make the following contributions.

- We propose a general method to detect the virtualized code section from a program execution trace. This challenging problem has been largely overlooked by the existing work which relies on an over-simplistic assumption.
- We design a new optimization method to simplify the execution trace based on boundary information. Our *multiple granularity symbolic execution* extends the capability of symbolic execution in analyzing malicious binary code.
- Our approach is capable of performing correctness testing to the deobfuscation results, which is rarely done by

the previous work. The source code are available at https://github.com/s3team/VMHunt.

2 BACKGROUND AND MOTIVATION

In this section, we first present the strength of code virtualization in complicating reverse engineering. Then we discuss the drawbacks of current deobfuscation work when handling threaded interpretation and selective virtualization. These limitations motivate us to develop a generic tool.

2.1 Obfuscation Strength of Code Virtualization

Code virtualization converts native binary code to bytecode written in a RISC-like virtual instruction set architecture (ISA), and a custom emulator is attached to interpret the bytecode at run time. Note that the new ISA can be generated randomly, and thus the bytecode vary greatly from one obfuscated version to another. The step 0 in Figure 1 shows an x86 instruction is translated to a virtual ISA in a stack architecture style. Static analysis of the custom bytecode is infeasible because the language specification of the new ISA is unknown. Figure 1 illustrates the classic way to implement code virtualization: decode-dispatch based interpretation. Step 1~4 form a central loop to decode, dispatch, and execute the bytecode. Many previous works perform dynamic analysis to identify these central loops and then find the mapping between each bytecode and its corresponding handler function [29, 30, 51, 55, 59]. The handler functions run on the native CPU and their semantics are equivalent to the bytecode. However, the handler functions are usually heavily obfuscated by other obfuscation schemes, such as hiding constant value by data encoding [16, 86], mutating instructions to syntactically different variants [1, 85], and control flow obfuscation [17, 74]. To get a relatively intelligible understanding of an obfuscated handler function, analysts have to perform heavy-weight optimization or simplification [9, 10, 22, 57].

2.2 Threaded Interpretation

Decode-dispatch interpretation is simple to develop, but the frequently-used indirect branches in this structure introduce extra overhead due to expensive mispredict penalty [25]. In addition, the decode-dispatch structure has become well-known to security researchers. Instead, commercial obfuscators have adopted an alternative structure of emulator, threaded interpretation [8, 24], to improve performance and complicate reverse engineering. Threaded interpretation removes the central decode-dispatch loop, which does not meet the assumption held by many deobfuscation tools. Figure 2 shows an *indirect threaded interpretation* [24] structure that appends the decode-dispatch routine within each handler function. The latest version of Code Virtualizer and Themida has applied indirect threaded interpretation to their code emulators. To further confuse decode-dispatch loop search, Code Virtualizer also attaches a fake loop to mimic the real one. For example, we virtualize four x86 instructions by Code Virtualizer's "tiger white" VM and generate three protected copies. The trace size of them ranges from $81,952 \sim 97,644$ lines of instructions (5 orders of magnitude explosion), and the fake decode-dispatch loop contributes about 90% of junk code. We have tested the three variants with Virtual

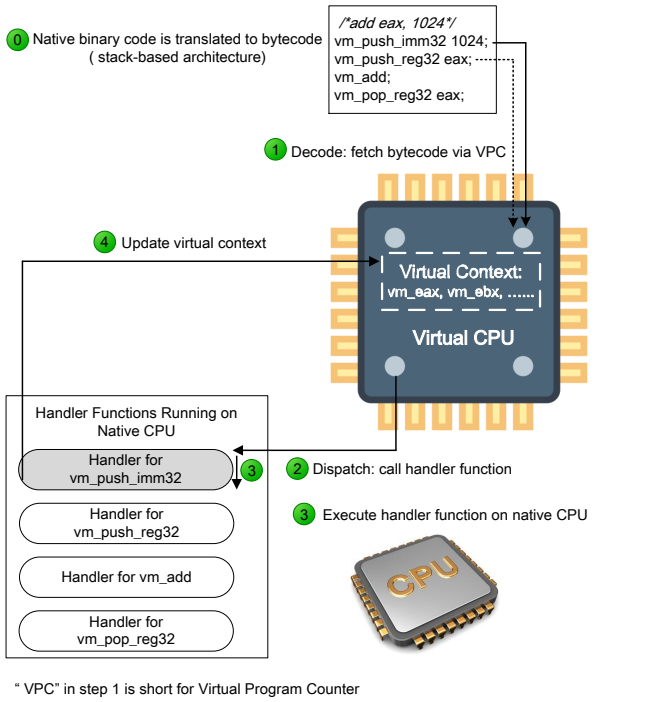


Figure 1: The classic code emulator structure: decode-dispatch based interpretation. Example: x86 Virtualizer [53], Tigress [15], and the demo versions of commercial software virtualization products [45, 46, 66, 69, 73].

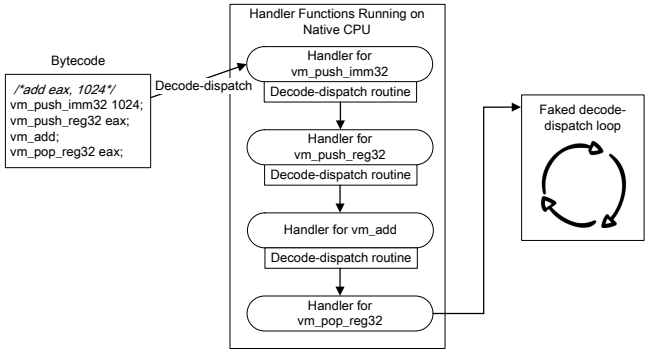


Figure 2: Indirect threaded interpretation, and a fake decode-dispatch loop is attached to mislead detection. Example: Code Virtualizer 2.2.2 [45] and Themida 2.4.6 [46].

Deobfuscator [51] and VMAttack [30], two open source tools that rely on the detection of decode-dispatch loop. Both of them fell into the trap and missed the real target. Compared to Code Virtualizer, VMProtect 3.1.1 follows the *direct threaded code* [48] style: the bytecode goes through a pre-decoding step so that each handler function's address is inlined into the bytecode. In this way, the bytecode can directly call the related handler function.

2.3 Virtualization-Obfuscated Malware

The strength of code virtualization comes with the problems of high performance penalty and poor compatibility. Whole program virtualization may not be an optimal option to cyber-criminals, as it can add new telltale signs such as high CPU usage and delay malware propagation. In practice, malware developers opt to virtualize the core parts of malware. For example, FinSpy malware applies custom code virtualization to protect command-and-control information [52, 70]; SpyEye uses VMProtect to protect the malware builder [65]; some malware samples choose to virtualize stolen code to evade user-level API hooking [31, 78]. However, none of the previous deobfuscation work studies the problem of partially-virtualized code, and many of them can only deal with the classic code emulator structure. Our approach will bridge this gap.

3 OVERVIEW

Figure 3 shows an overview of VMHunt's workflow, which contains three key components.

(1) *Virtualized Snippet Boundary Detection.* Given a virtualized program, we first run it to record an execution trace. Then we identify the virtualized snippet boundary in the trace by detecting *context switch instructions*. Those instructions switch context between the native environment and the virtualized environment.
(2) *Virtualized Kernel Extraction.* After the boundary is detected, we analyze and extract the *kernel* of the virtualized snippet. The kernel refers to instructions in the virtualized snippet that affect native program environment. It reveals the semantics of the virtualized snippet.
(3) *Multiple Granularity Symbolic Execution.* We propose a new symbolic execution called "multiple granularity symbolic execution" to simplify virtualized snippets. Our method represents the semantics of kernel virtualized code as concise symbolic formulas.

4 VIRTUALIZED TRACE BOUNDARY DETECTION

4.1 Trace Logging

VMHunt's trace log component is based on Intel Pin [38], a dynamic binary instrumentation framework. The trace logger can record all the instructions executed during runtime except those inside system calls. In addition to the instructions, a trace also includes plenty of runtime information. Overall, the following information is recorded in a trace.

(1) The memory address of every instruction
(2) The instruction name (opcode), which describes the operation that the instruction is performing
(3) The source and destination operands. Usually there are three types of operands: immediate value, register, or memory address
(4) Runtime information, including the content in all registers and memory accessing addresses

4.2 Context Switch Instructions

One of our major observation is that code virtualization is typically applied to some sensitive code sections, and the rest parts still run as native code. Therefore, during the execution of a partially-virtualized program, it will switch between the native environment and the virtualized environment. Figure 4 presents a trace example showing the execution of a virtualized program. Before line 1, the program is running in the native context. The instructions from line 2 to 9 save the native context by pushing all general registers and the flag register to stack. After that, the jump instruction at line 10 transfers the execution to the virtualized environment. From line 11 to 15, the program is running inside the virtualized context. After reaching the end of the virtualized part at line 15, the instructions from line 16 to line 23 restore the context by popping values from memory to registers. Finally, the jump instruction at line 24 transfers the execution back to the native program and continues the execution.

In this paper, we define the instructions that save or restore the context between the native and virtualized environment as *context switch instructions*. Specifically, there are two categories of context switch instructions: context *saving* instructions and context *restoring* instructions. Context saving instructions save all registers to memory (typically stack), such as the instructions from line 2 to 9 in Figure 4. Similarly, context restoring instructions restore all registers from memory (typically also stack), such as the instructions from line 16 to 23 in Figure 4. The term "context" in this paper refers to the content in all registers.

Two VM Architectures: Stack vs. Register. A long-running discussion in VM design is whether an interpreter should be implemented via stack-based architecture[1] or register-based architecture, because these two designs have their own pros and cons [21, 60]. We wish to emphasize here that context switch instructions are independent of any specific VM architecture. For example, Code Virtualizer [45] provides multiple VMs including both stack-based VM and register-based VM, and we find context switch instructions are very common among different VMs. Since all partially-virtualized programs include context switch instructions, they are significant symbols for the beginning or end of virtualized snippets. Therefore, identification of these context switch instructions is the first step towards the detection of virtualized snippets.

Apparently, for the ideal case shown in Figure 4, identifying context switch instructions is fairly straightforward. One intuitive solution is using a pattern match method to match instruction sequences that push all registers to stack or pop them back. However, according to our observation, virtualizer developers have already taken advantage of obfuscation methods such as code mutation [1, 85] to hide context switch instructions. Figure 5(a) shows an example of obfuscated context saving instructions in Code Virtualizer, and we have trimmed and simplified the example for better presentation. In Figure 5(a), the instructions from line 4 to 15 is actually an obfuscated version of `push edi`. In order to detect the context switch instructions within an obfuscated execution trace, which usually contains millions of instructions, we propose a three-step method to effectively identify them. The three steps are normalization, simplification, and clustering.

[1]The step 0 in Figure 1 is an example of stack-based architecture.

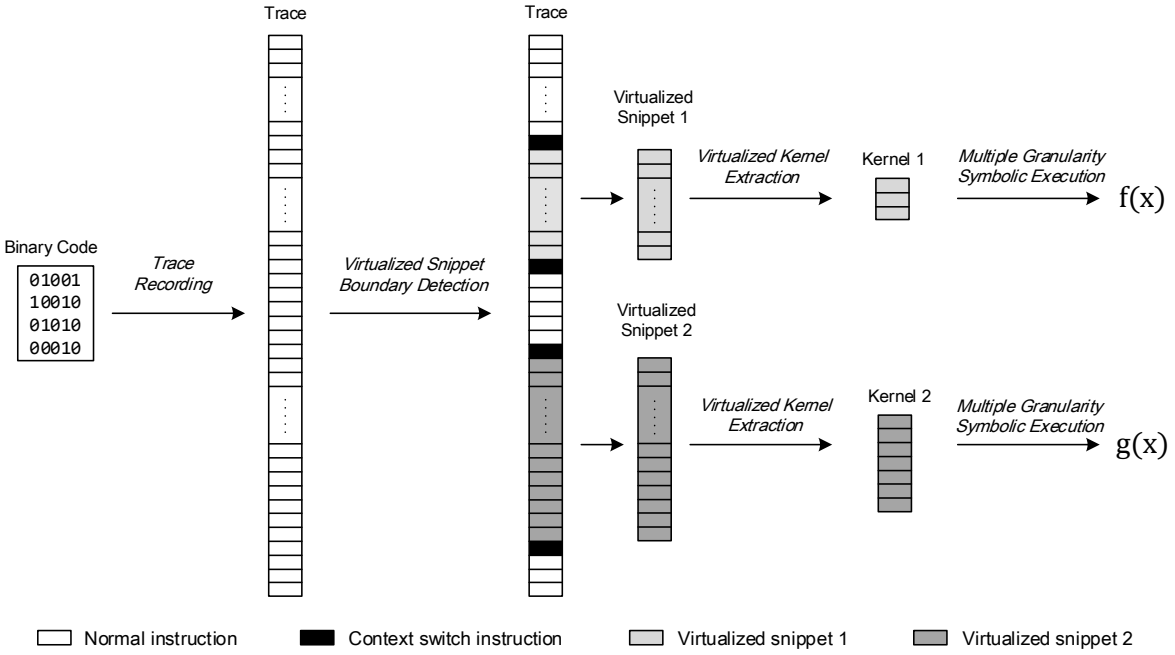


Figure 3: An overview of VMHunt's workflow. Virtualized snippet 1 & 2 represent two different VM architectures. The words in italics represents VMHunt's key components.

```
1  ...             // native program execution
2  push  edi       // context saving
3  push  esi
4  push  ebx
5  push  edx
6  push  ebp
7  push  ecx
8  push  eax
9  pushfd
10 jmp   0x1234
11 ...             // virtualized snippet begin
12 mov   ...
13 add   ...
14 xor   ...
15 ...             // virtualized snippet end
16 popfd
17 pop   eax       // context restoring
18 pop   ecx
19 pop   ebp
20 pop   edx
21 pop   ebx
22 pop   esi
23 pop   edi
24 jmp   0x8048123
25 ...             // continue native program
26 ...             // execution
```

Figure 4: An execution trace showing context switching in a virtualized program.

(1) *Normalization.* We normalize all the data transfer instructions (`push`, `pop`, `xchg` etc.) using the `mov` operation. Normalization can resist instruction replacement obfuscation and reveal more simplification opportunities.

(2) *Simplification.* Similar to the previous work in automatic binary deobfuscation [29], our simplification involves two main components: a peephole optimizer and a data flow analyzer. The peephole optimizer is based on pattern matching rules. It removes redundant instructions, e.g., store and load the same data between registers and memory. The data flow analyzer records the def-use information of the operands, and then perform constant propagation and dead code elimination.

(3) *Clustering.* Based on the normalized and simplified trace, we build an instruction dependency graph. Our goal is to cluster the instructions that load data from memory to registers together without affecting the dependency. Similarly, instructions that store data from registers to memory are also clustered together. If one cluster include operations on all registers, the instructions in the cluster is considered as context switch instructions. For example, if one cluster includes several `mov` instructions from all registers to memory, we regard them as context saving instructions.

We use the example in Figure 5 to show the procedure of detecting context switch instructions in an obfuscated trace. Figure 5(a) is the trace containing obfuscated context switch instructions. For simplicity, we place the `push` instructions for `esi`, `ecx` and `edx` at the first line. They are handled in the same way as the `push` instructions at line 2 and 3. Figure 5(b) is the result of normalization. All the `push` instructions are normalized to a `mov` and a `sub` instruction as shown from line 2 to 5 in Figure 5(b). The blank lines are not inserted by the normalization procedure and are for format only. Next, the simplification procedure works on the normalized trace.

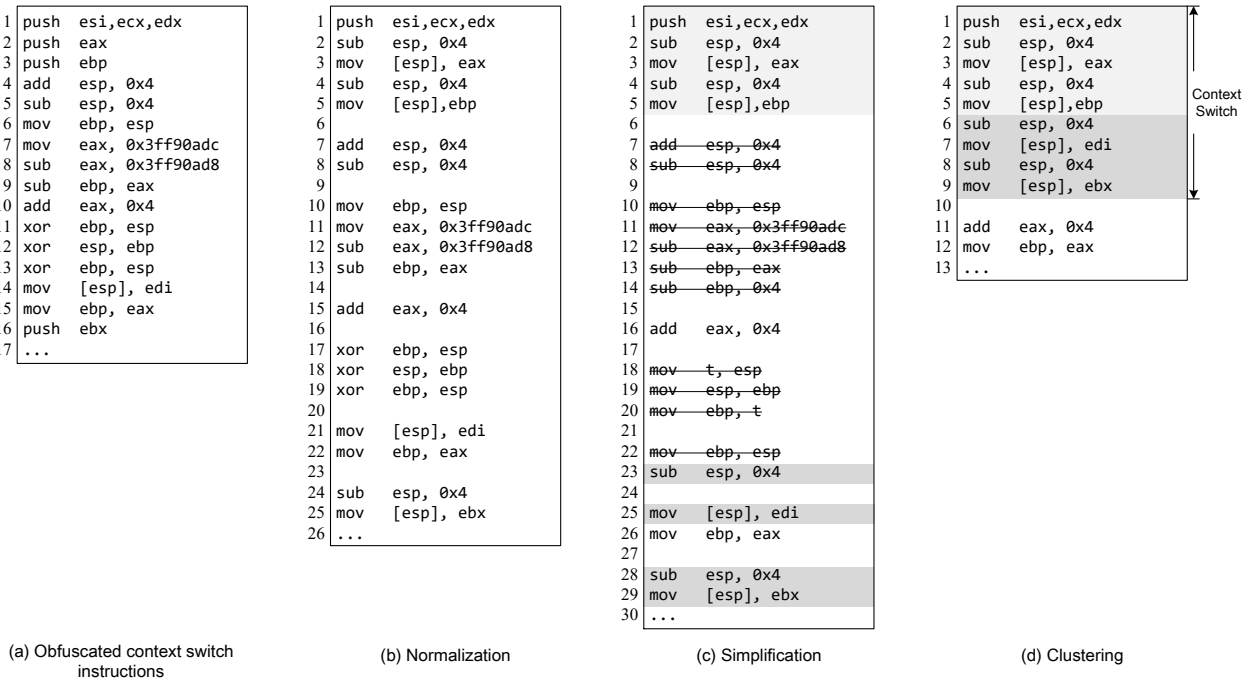


Figure 5: Detection of context switch instructions in an obfuscated program trace. The instructions in the gray area in (d) are a cluster of context switch instructions. The color marks where the instructions are clustered.

The peephole optimizer removes the redundant instructions at line 7 and 8 in Figure 5(b). It also replaces the three `xor` instructions from line 17 to 19 in Figure 5(b) with three instructions exchanging data between `esp` and `ebp` as shown from line 18 to 20 in Figure 5(c). Then the data flow analyzer simplifies the instructions at line 10-14 and 18-20 to instructions at line 22-23 in Figure 5(c). The instruction at line 22 is deleted as dead code because the value produced is never used. The final simplification result is shown in Figure 5(c). After building the instruction dependency graph, the clustering procedure finds the instructions at line 16 and 26 in Figure 5(c) has no dependency on the instructions below them (labeled as grey). Therefore, the data saving and stack changing instructions at line 23, 25, 28, and 29 can be clustered with instructions from line 1 to 5. Finally, the clustering result is shown in Figure 5(d)'s gray part. It includes all of the instructions related to context saving, so we can recognize these instructions as context switch instructions.

4.3 Pairing Context Switch Instructions

The next step is to pair the context saving instructions with the restore instructions, so that we can further identify the virtualized snippet between them. We use two heuristics to guide the pairing process: *stack depth* and *execution transfer instruction*. In this paper, stack depth denotes the value of the stack pointer register, for example, the `esp` register in 32-bit x86 architecture. In the scenario of switching context between native and virtualized program, the stack depth of the context saving instructions should be the *same* as the context restoring instructions. Suppose one program is switching from native context to virtualization execution at point A with the stack depth n. When the execution of virtualized part is finished, it reaches program point E. It will restore the context of normal program and continue execution from E. To ensure the accuracy of the execution, the stack depth at E should be the same as that before entering the virtualized part, which means the stack depth at E is n as well.

When pairing the context switch instructions, we only pair the context saving and restoring instructions in the same stack depth. In this way, we overcome the shortcomings of blindly recursive pairing. First, only context switch instructions in the same stack depth is likely to be the start or end point of a virtualized program. On the other hand, the context switch instructions in one stack depth are separated from those in other stack depths. Therefore, even if the context switch instructions in one stack depth is mistakenly paired, it will not affect the instructions in other stack depths.

The other heuristic for pairing context switch instructions is that, virtualization related context switch instructions usually come with a *control flow transfer instruction*. It switches the execution between the normal program and the VM, such as the jump instructions at line 10 and 24 in Figure 4. Therefore, we only select those context switch instructions that are followed by a control flow transferring instruction, which could be any instruction used to change the execution flow, such as `jmp, call, ret`.

5 EXTRACTION OF VIRTUALIZED KERNEL

The virtualized snippets collected from the last step are still too large for analysis. In addition, modern virtualization tools would mix other obfuscation techniques with virtualization so as to further increase the obfuscation strength. Previous work [19, 59, 83] already discussed some methods to deobfuscate a virtualized code, but they

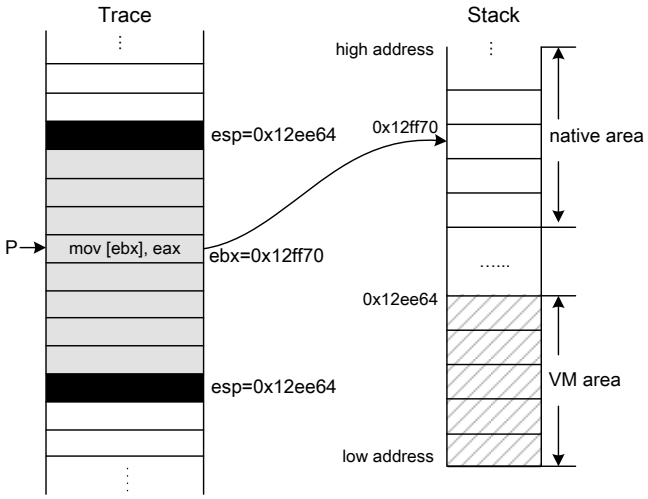


Figure 6: An example showing an instruction with global behavior. It modifies the native area in the program stack.

did not take the boundary information into consideration. In this section, we will discuss how the boundary information is used to extract the core part of virtualized snippets.

In general, our goal is to understand the behavior of a virtualized snippet. The behavior can be categorized into two types: *local* behavior and *global* behavior. *Local* behavior means its effect is restricted in the virtualized context without affecting the native context. An instruction with *global* behavior means it accesses or changes the outside native context. For example, one instruction has global behavior if it saves data to the outside context. When analyzing a virtualized snippet, those instructions with global behaviors are crucial because they reveal the real function of the snippet. In other words, the global behavior instructions are the interface between the virtualized context and the normal context. The virtualized code has to utilize these instructions with global behavior to read in parameters and write the result to the native context. Therefore, the global behavior instructions and the instructions that are related to them constitute the core part of a virtualized snippet. We call those instructions the "Kernel" of a virtualized snippet. Our key insight is that the kernel can represent the semantics of a virtualized snippet. We only need to focus on the kernel when analyzing a virtualized snippet. In order to extract the kernel, we first identify the instructions with global behavior in a virtualized snippet, and then slice the trace based on those instructions. The following part describes the extraction steps in detail.

First, we start by defining two terms regarding two different stack memory areas: the *native area* and the *VM area*. We use the stack depth (discussed in Section 4.3) as a delimiter to separate VM stack area from the stack area used by the native program. Note that the native area does not intrude the VM area, because they are typically very far from each other in the stack memory. In practice, if the VM stack is mixed with the native program stack, the whole virtualized program will become unstable and easy to crash. It also increases the complexity of designing the VM memory. Therefore, all commercial obfuscators avoid mixing them together.

With the definition of native area and VM area of stack, we can go further to define the global behavior of an instruction in a virtualized snippet. In general, the global behavior includes native context access or modification. One of the common scenarios of global behavior is an instruction modifies the content in native stack, and Figure 6 illustrates such an example. The program execution reaches point P in the trace and P is inside the virtualized snippet. The stack depth at the context switch instruction is `0x12ee64`. The stack view at the point P is shown in the right part. Assuming the stack is growing top-down, the section above address `12ee64` is the native area and the section below is the VM area. The instruction at program point P is writing data from register `eax` into memory address `[ebx]`. Since the content of `ebx` is `0x12ff70`, which means the target address is located in the native area, the instruction at P has global behavior.

In addition to explicitly modifying the native area content, another implicit way of changing the native area is to modify the stack memory content that will be swapped out by the context restoring instructions (e.g., line 16 to 23 in Figure 4). Because the content is popped to registers and will be used by the native execution, modifying that memory content will affect the native execution as well. Therefore, we define the following two types of instructions inside a virtualized snippet as global behavior instructions. The kernel of a virtualized snippet consists of the instructions that are related to global behavior instructions.

(1) The instruction writes data to the native area of stack. It is used for directly passing execution result of virtualized snippet to the native area.
(2) The instruction saves data to the stack memory that will be swapped to registers by context switch. It is used for indirectly passing the execution result of virtualized code to native program.

We run a backward slicing to extract the kernel, which contains all the instructions related to the global behavior instructions. In particular, we apply BinSim's enhanced backward slicing algorithm [40], which is able to handle many complicated issues when performing slicing on binary code (e.g., implicit branch logic). The sliced trace includes all of the necessary instructions that contribute to computing the outputs from inputs, so it represents the real semantics of the virtualized part. The slicing can significantly remove unnecessary instructions in the trace. The evaluation result in Section 8 shows that the size of the kernel only takes up about 1% of the total instructions of the initial virtualized snippet.

6 MULTIPLE GRANULARITY SYMBOLIC EXECUTION

To extract the semantics of the virtualized code, we want to get a formula from the virtualized kernel by symbolic execution. By comparing formulas from the original and obfuscated code, we can further verify whether our deobfuscation is a semantically equivalent translation. This correctness testing is never done by previous work. However, another challenge rears its head. Modern virtualized code usually come with lots of bit-wise operations. For example, the VM byte code could be 8 bits or 16 bits, or even 9 bits [79]. Also, data obfuscation can generate variables in different granularities. In this scenario, although traditional symbolic execution with a fixed

atomic granularity (such as 32 or 64 bits) is still able to correctly generate formulas, it fails to optimize the formulas which contain symbols in different granularity. Traditional symbolic execution only literally translate the instructions to formulas, and let solvers to optimize and verify the formulas. These unoptimized formulas are typically very large and include lots of redundant variables, resulting in too much time for a solver to solve them.

On the other hand, the most fine-grained symbolic execution such as bit-precise symbolic execution is able to remove redundant bit-wise formulas. However, it could otherwise become a performance overkill. Since bit-precise symbolic execution breaks the granularity of all variables into bit level, the resulting boolean formulas could be too large for analysis and far from intelligible as well. Therefore, fine-grained symbolic execution such as bit-precise symbolic execution is not appropriate for the simplification of virtualized snippet. In order to optimize formulas containing multiple-granularity variables, we propose a novel symbolic execution called "multiple granularity symbolic execution", in which the length of symbols is not fixed during the symbolic execution. Compared to the traditional single granularity symbolic execution, our method can perfectly optimize formulas during symbolic execution and generate neat formulas, which are easily handled by theorem solvers.

Basically, multiple granularity symbolic execution tries to "interpret" the semantics of every instruction in terms of different granularities, rather than only "translate" instructions to formulas and let the solver to handle the semantics. The new features of multiple granularity symbolic execution are elaborated as follows.

(1) Maintain runtime status of registers during symbolic execution, including the size and location of symbols and concrete values in every register.
(2) Interpret the effect of each instruction, e.g., shift left/right the symbols in a register. Instead of translating the instruction to a `shl` or `shr` in the output formula, multiple granularity symbolic execution updates the content in registers with the shifted result.
(3) Remove redundant symbols if they become concrete values after the interpretation of the instruction, e.g., `and sym, 0x00` when `sym` is an 8-bit symbol. The result is a concrete value `0x00` and the symbol `sym` becomes redundant after the interpretation, so it will be removed.

Multiple granularity symbolic execution is designed for optimizing formulas containing variables on different granularity during symbolic execution. In single granularity symbolic execution, the granularity of a symbol is fixed when it is claimed, for example, 32 bits or 64 bits, or only 1 bit. Similarly, a concrete value's granularity is also fixed. Users are not able to claim a 32-bit value, in which the first 10 bits is one symbol and the last 22 bits is a concrete value. In our multiple granularity symbolic execution, the key idea is that we provide the capacity of creating a half-symbolic and half-concrete value. With this strength, our symbolic execution can naturally handle the bit-level operations and reduce the formula size. Figure 7(a) shows an example. In the rest of this section, we will use this example to elaborate how multiple granularity symbolic execution works.

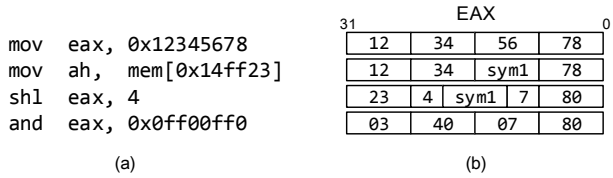


Figure 7: An example showing the states of multiple granularity symbolic execution.

The snippet includes four instructions. The first instruction moves a concrete value `0x12345678` to the 32-bit register `EAX`. The second instruction loads a 8-bit symbolic value from memory to the 8-bit register `AH`. The third instruction shifts the content in `EAX` to left by 4 bits. The last instruction performs conjunction on the value in `EAX` and a concrete value `0x0ff00ff0`. Figure 7(b) shows the content of `EAX` after each instruction. The key insight here is that, the symbolic value `sym1` is eliminated during the execution and the result is a concrete value. However, single granularity symbolic execution usually ignore this fact and still generate a symbolic value for the result, which leads to many redundant variables in formulas. Our multiple granularity symbolic execution can catch the fact and correctly generate a concrete value for the result. In the following part, we will compare the execution procedure of the two types of symbolic execution to show how our method works. As the first step, we elaborate the symbols that will be used for describing the symbolic execution procedure.

Definition 6.1. In symbolic execution, there are two types of value: concrete value and symbolic value. We use C_n^m to represent a concrete value in symbolic execution. m is the length of the value. n is the unique id of the value. Similarly, S_n^m represents a symbolic value. For example, C_2^8 represents a 8-bit concrete value and S_3^{16} represents a 16-bit symbolic value.

Definition 6.2. We use $[\text{Value}_1, \text{Value}_2, \ldots, \text{Value}_n]$ to represent the concatenation of the n values. For example, $[C_1^8, S_2^8, C_3^8]$ means a concatenation of the three 8-bit values.

Definition 6.3. The symbol | is used for binding values. For example, $[S_1^8, C_2^8]|_{C_2^8=\texttt{0x23}}$ means a concatenation of a symbolic value S_1^8 and a concrete value C_2^8. The concrete value C_2^8 is `0x23`. In another word, C_2^8 is bound to `0x23`.

Given the definitions above, we first simulate the process of single granularity symbolic execution. In practice, the granularity of most symbolic execution engines is one byte, which means the shortest length of a value is 8 bits. So here we use 8 bits as the granularity in this example. Every line is the execution result of one instruction in Figure 7(a).

$$[C_4^8, C_3^8, C_2^8, C_1^8] \;|_{C_4^8=\texttt{0x12}, C_3^8=\texttt{0x34}, C_2^8=\texttt{0x56}, C_1^8=\texttt{0x78}} \quad (1)$$

$$[C_4^8, C_3^8, S_1^8, C_1^8] \;|_{C_4^8=\texttt{0x12}, C_3^8=\texttt{0x34}, C_1^8=\texttt{0x78}} \quad (2)$$

$$S_3^{32} = \texttt{shl}(S_2^{32}, C_5^8) \;|_{S_2^{32}=[C_4^8, C_3^8, S_1^8, C_1^8], C_5^8=\texttt{0x4}} \quad (3)$$

$$S_4^{32} = \texttt{and}(S_3^{32}, C_6^{32})|_{C_6^{32}=\texttt{0x0ff00f0}} \quad (4)$$

In step (1), four 8-bit concrete values are concatenated together to show the content of EAX. In step (2), the 8-bit symbolic value S_1^8 is created to represent the value in AH. However, the 8-bit symbolic execution do not have the capability to represent a 4-bit symbolic value. Therefore, in step (3), it concatenates the four 8-bit values and treats them as a 32-bit symbolic value S_3^{32}. In step (4), S_3^{32} is used for calculating the conjunction.

From the simulation above, we can see that the result of single granularity symbolic execution is a 32-bit symbolic value. However, Figure 7(b) shows that the execution result should be a concrete value. Therefore, single granularity symbolic execution create a redundant symbolic value in the formula. Although the theorem solver in the following step can prove that this symbolic value is actually a concrete value, these redundant values unnecessarily increase the formula size and add more burden to the solver.

The reason that single granularity symbolic execution misses the optimization opportunity is that it lacks the capacity of handling values on multiple granularity. Next we will show how our multiple granularity symbolic execution is able to solve this problem. The execution procedure is simulated as follows. First, we create a 32-bit concrete value.

$$C_1^{32}|_{C_1^{32}=\texttt{0x12345678}}$$

The second instruction sets AH to an 8-bit symbolic value. Since the granularity is flexible, our method can directly set the corresponding part to a symbolic value, leaving the remaining bits untouched as concrete values. In our example, it splits the 32-bit concrete value into one 16-bit value and two 8-bit values, and then replaces one 8-bit concrete value with a 8-bit symbolic value S_1^8. The result is shown as below.

$$[C_2^{16}, S_1^8, C_3^8]|_{C_2^{16}=\texttt{0x1234}, C_3^8=\texttt{0x78}}$$

The third instruction shift $[C_2^{16}, S_1^8, C_3^8]$ to the left by 4 bits. In multiple granularity symbolic execution, bit-level granularity is exposed to all bitwise operations. Therefore, our method is able to precisely interpret the semantics of bitwise operations on bit level. The symbolic execution result of the shift operation is shown as follows. It is an concatenation of one 12-bit concrete value, one 8-bit symbolic value and another 12-bit concrete value.

$$[C_4^{12}, S_1^8, C_5^{12}]|_{C_4^{12}=\texttt{0x234}, C_5^{12}=\texttt{0x780}}$$

The last instruction perform the conjunction operation. Similarly, Multiple granularity symbolic execution engine accurately execute the conjunction instruction on bit level as follows.

$$[C_4^{12} \wedge \texttt{0x0ff}, S_1^8 \wedge \texttt{0x00}, C_5^{12} \wedge \texttt{0xff0}]|_{C_4^{12}=\texttt{0x234}, C_5^{12}=\texttt{0x780}}$$

Therefore, the final result is a concatenation of three concrete values.

$$[C_6^{12}, C_7^8, C_8^{12}]|_{C_6^{12}=\texttt{0x034}, C_7^8=\texttt{0x00}, C_8^{12}=\texttt{0x780}}$$

The three concrete values can be further merged to one 32-bit concrete value. This is the same result as that shown in Figure 7(b).

$$C_9^{32}|_{C_9^{32}=\texttt{0x03400780}}$$

Advance over state-of-the-art work. Multiple granularity symbolic execution is a new variant of symbolic execution to summarize the semantics of the extracted code, and it balances the accuracy and performance between fixed bit-level symbolic execution and traditional symbolic execution. By contrast, extending existing work to achieve the same goal is difficult, if not impossible. In summary, our design offers two competitive advantages.

(1) *Fine-grained Analysis.* Multiple granularity symbolic execution accurately interprets the semantics of bitwise operations on bit level during execution. The fine-grained information exposes optimization opportunities for eliminating redundant symbolic values.
(2) *Flexibility.* The multiple granularity symbolic execution engine is free to split and merge values without granularity restriction. This feature gives multiple granularity symbolic execution the capacity of operating on different granularity levels, so that it can perform fine-grained analysis but still generate concise formulas.

7 IMPLEMENTATION

We build an open source tool called *VMHunt* as the prototype of our idea. The trace logger is written in C++ based on Intel's Pin DBI framework [38] (version 2.13). Symbolic execution on the binary code has appealing applications in security analysis, and many options [11, 14, 20, 22, 58, 61, 64] are available in the arsenal. However, the fixed data type design in existing symbolic execution engines obstructs the implementation of multiple-grained symbolic execution. We have to redesign two fundamental components in the symbolic execution engine: 1) a new data structure to support multi-granularity data types; 2) a new symbolic execution rule to decide an instruction should be translated to formula or interpreted. It motivates us to develop our own symbolic execution engine. We design an intermediate representation (IR), which can effectively encode symbolic and concrete values on different granularities. Based on this new feature, our symbolic execution engine efficiently interprets the behavior of instructions and translates it to concise formulas. Specifically, VMHunt consists of several components, including the multiple granularity symbolic execution engine, a parser for lifting a trace to the IR, the virtualized snippet boundary detector, a forward/backward slicer, a peephole optimizer to revert the effects of instruction-level obfuscation, and utilities for control flow graph generation. The whole tool chain includes 17, 192 lines of C++ code and 341 lines of Perl code.

8 EVALUATION

In this section, we evaluate VMHunt from two aspects: *effectiveness* and *performance*. Particularly, we design and run experiments to answer the research questions (RQs) as follows.

(1) **RQ1:** Is VMHunt able to correctly detect the virtualized snippet boundary inside a virtualized program trace? (*effectiveness*)
(2) **RQ2:** Is VMHunt able to effectively extract and simplify the virtualized kernel? (*effectiveness*)
(3) **RQ3:** How many false positives can VMHunt produce? (*effectiveness*)
(4) **RQ4:** How much overhead does VMHunt introduce? (*performance*)

As the response to RQ1, we first apply modern commercial virtualizers to several real open source programs and then use VMHunt

to detect the virtualized snippets. In RQ2, we compare the size of kernel with the size of virtualized snippet. We use a theorem prover, STP [27], to check the equivalence of the kernel and the original trace (i.e., correctness testing). We run VMHunt on malware samples and provide a case study about a virtualized ransomware to answer RQ1 and RQ2. In response to RQ3, we run benign programs without virtualization to check the false positives. As the answer to RQ4, we report the performance of the main components in VMHunt, including the tracer, boundary detector, and symbolic execution engine.

8.1 Open Source Programs

Typically, virtualization is used for protecting a piece of sensitive snippet in a program. So first we evaluate VMHunt in this common scenario. We select several open source programs as the test bed. Next, we apply modern virtualization obfuscators to a snippet in the program. After that, we use VMHunt to discover the virtualized snippet and simplify it.

8.1.1 Testbed Programs. Our testbed is comprised of programs from several open source projects. We choose them based on the following facts. First, they are widely used open source programs in the real world. Second, they are representative tools from different areas. Lastly, they inherently include a main loop that read data and process it. For example, a web server contains a loop to dispatch different inquire packages to the corresponding handlers based on the package type. The main loop's behavior is similar to a virtual machine's dispatch-handler behavior. This behaviors can help us check the false positives that VMHunt produces. The programs in our testbed are grep-2.21, bzip2-1.0.6, md5sum-8.24, AES in OpenSSL-1.1.0-pre3, thttpd-2.26, and sqlite-2.26. The CPU and memory of our testbed machine is Intel Core i7-3770 processor and 8GB, with Ubuntu Linux 14.04 installed.

8.1.2 Virtualizer and Sensitive Area. We virtualize the above testbed programs with several modern virtualization obfuscation tools and then use VMHunt to detect and simplify the virtualized snippet. The virtualization tools are Code Virtualizer [45], Themida [46], VMProtect [73] and EXEcryptor [66]. We adopt the most recent released version of each tool so as to evaluate VMHunt against the state-of-the-art virtualization techniques[2]. All those virtualization tools provide the capability to let users select a piece of sensitive area in a program to be virtualized. The virtualization tool will convert that area to virtual instructions that can only be understood by an internal virtual machine. The remaining part of that program will be untouched.

In order to evaluate VMHunt in the real world scenario, we implement a trial/registration scheme which has two virtualized areas in each of the test bed programs. The details of the trial/registration is shown in the Appendix A. After that we use VMHunt to detect the boundaries of the virtualized areas. The result shows that VMHunt correctly identifies the virtualized snippets in all testbed programs. VMHunt is also able to extract the kernel of each virtualized snippet. We compare the number of lines of the total trace, virtualized snippets and kernels in Table 1. For simplicity, we only present the evaluation data from Code Virtualizer [45]. The data from VMProtect [73] and EXECryptor [66] are similar. Themida [46] shares same VMs with Code Virtualizer, so the evaluation data is also the same. According to Table 1, the virtualized snippet identified by VMHunt is about 10% of the whole trace size. The kernel of a virtualized snippet is about 10^{-4} of the whole trace size. The result proves that VMHunt can significantly reduce the number of instructions for future analysis.

Next, we run multiple granularity symbolic execution on each kernel of the virtualized snippets. The symbolic execution generates a simplified formula representing the semantics of the kernel. To check whether the formula is equivalent to the original program, we run symbolic execution on the trace of the original program without virtualization. After that we use STP [27] to check whether the formula from VMHunt's output is equivalent to the unobfuscated formula. The experiment result shows that all formulas generated by VMHunt is equivalent to the original formula before obfuscation. Compared to other deobfuscation work, VMHunt is the first one that can verify the correctness of simplification result.

In order to evaluate the multiple granularity symbolic execution in VMHunt, we compare it with two single granularity symbolic execution engines. One is on byte-level (8 bits) and the other one is on bit-level (1 bit). We remove the multiple granularity component from our symbolic execution engine and modify it to a byte-level and bit-level symbolic execution engine separately. They are used for the comparative evaluation with VMHunt. We run the two single granularity symbolic execution engines on the virtualized kernels and compare the formulas with those generated by VMHunt. The result is shown in Table 2. It shows that the size and number of variables from formulas generated by multiple granularity symbolic execution is significantly less than those from byte or bit level symbolic execution engines. Moreover, we also compare the time that a solver take to solve the formula. The formula generated by multiple granularity symbolic execution can be solved about 10X faster than that from byte-level symbolic execution and 20X faster than bit-level symbolic execution. The result proves that the multiple granularity symbolic execution in VMHunt can produce concise and efficient formulas, especially for bitwise operations.

8.2 Multiple VMs Virtualization

Some modern virtualization tools, such as Code Virtualizer [45] and Themida [46], come with multiple custom VMs. Those custom VMs are designed using different architectures. Users can apply different VMs to different sensitive areas in the same program. As a result, the execution trace of that program contains multiple virtualized snippets of different VMs. In this scenario, reverse engineering one sensitive area provides very little information for cracking other sensitive areas. It raises a strong challenge to existing deobfuscation methods and we have not found a direct response to this challenge in the previous work.

We are curious about VMHunt's performance in the scenario of multiple VM virtualization. We conduct an experiment to verify whether VMHunt is able to detect and simplify all virtualized snippets of different VMs in a program. We adopt Code Virtualizer [45] in this experiment, and it shares the same custom VM engines with Themida [46]. The name of each VM is formed by an animal name

[2]We purchased the professional editions of all testing obfuscators. They are still the latest versions until 05/09/18.

Table 1: The number of instructions of the whole trace, virtualized snippets, and kernels in all testbed programs. T means the whole trace. S1 and S2 are the two virtual snippets in the trace. K1 and K2 are the kernels in S1 and S2 respectively.

Programs	T	S1	S2	S1+S2	K1	K2	K1+K2	(S1+S2)/T(%)	(K1+K2)/T(10^{-4})
grep	1,072,446	130,329	168,857	299,186	552	1,061	1,613	24.6	15.0
bzip2	1,422,428	133,272	153,537	286,809	774	1,444	2,218	20.2	15.6
aes	2,479,948	124,793	156,019	280,812	837	1,173	2,010	11.3	8.1
md5sum	2,309,826	134,320	168,163	302,483	604	1,271	1,875	13.1	8.1
thttpd	3,680,610	117,435	155,262	272,697	677	1,389	2,066	7.4	5.6
sqlite	4,716,883	146,177	161,073	307,250	820	1,465	2,285	6.5	4.8

Table 2: Comparison of the formulas generated by bit-level, byte-level and multiple granularity symbolic execution. The formula size is measured by number of lines. The second metric is the number of variables. The third metric is the solving time measured by seconds. - means timeout after 1800 **seconds.**

SE	Metrics	grep	bzip2	aes	md5sum	thttp	sqlite
byte	size	671	459	674	801	792	997
	var #	1289	2071	3215	4318	4730	6103
	time	90	105	150	152	144	183
bit	size	7992	5205	5310	9134	6840	10289
	var #	25110	41947	69827	87638	80592	13609
	time	383	486	532	517	793	-
MG	size	71	128	218	239	291	348
	var #	408	558	544	673	804	930
	time	12	16	13	14	20	23

plus a color, such as "lion black", "tiger red." One animal name refers to a custom VM architecture and the colors means different variants of that that architecture. Deeper color means the variant has more virtualization methods applied in that VM architect. In this experiment, we apply two significantly different VMs, "tiger white" and "fish black" to virtualize the two sensitive areas as shown in Figure 9. After that we repeat the experiment in the last section. The result shows that all virtualized snippets in different VMs are correctly detected and simplified. The size of different sections is shown in Table 3. Particularly, by comparison of the S2 column in Table 3 with the S2 column in Table 1, we can see that the "fish black" VM is more complicated than "tiger white." However, the size of the kernel is still similar. It proves that VMHunt can extract the core semantics of the virtualized snippet.

Another way of applying multiple-VM virtualization is nested VM [83], which means, apply another virtualization to an already virtualized code. We also evaluate VMHunt in the nested VM scenario. We use grep as the testbed and apply the tiger-white VM twice. In practice, we observe that virtualized snippet after the first round of virtualization is significantly larger than the original snippet as shown in Table 3. Therefore, if the second round of virtualization is applied directly to the virtualized snippet, it will cause performance problem. Besides, the virtualized snippet also contains program structures which are not suitable for being virtualized again as mentioned in Section 1. In our evaluation, we select one block of instructions after in the first-round virtualization. That block does not include any program structures that could lead to wrong virtualization. Then we apply another round of virtualization to the block to produce the nested-virtualized grep program.

In our evaluation, we run VMHunt on the nested-virtualized grep. First, VMHunt correctly identify two virtualized snippets. Different from the previous experiment, those two virtualized snippets are nested. We first apply VMHunt to extract the kernel of the inner virtualized snippet and replace it with the kernel. After that, we run VMHunt again to process the outer virtualized snippet. The final result shows that VMHunt is able to correctly handle nested-VMs.

8.3 Malware Samples

In order to evaluate VMHunt in a malicious scenario in practice, we collect malware samples from Virustotal[3] and some other forums. These samples cover different malware categories, such as botnet, virus, and ransomware. All samples are already known as being virtualized. We run VMHunt on these malware to detect and simplify the virtualized snippet. Table 4 shows the evaluation result. VMHunt successfully detects all virtualized snippets in all the 10 samples. We also manually verify the extracted snippets are real virtualized snippet. A detailed case study of the ransomware sample, tears, is presented in the next section. Our observation is that malware developers only apply virtualization obfuscation to a small piece of code, which is typically a sensitive area that can be detected by the anti-virus software.

8.4 A Ransomware Case Study

In this section, we present a case study about applying VMHunt in a real-world ransomware called "tears". We download the ransomware sample from VirusTotal. The ransomware encrypts victims' files by AES encryption and then ask the victims to pay a ransom for decryption of their files. Based on the description on VirusTotal, the core part of the sample is protected by virtualization techniques. We set up a Cuckoo sandbox[4] to run the ransomware and record an execution trace. Then we use VMHunt to analyze the virtualized snippets inside the trace. The size of the whole trace, virtualized snippets and the kernels is shown as the last row in Table 4.

VMHunt identifies one virtualized snippet inside the execution of the ransomware. After extraction and multiple granularity symbolic execution of the kernel, we discover that the virtualized part is actually the procedure of key generation. In fact, the AES encryption key is generated based on the time when the ransomware

[3]https://www.virustotal.com
[4]https://cuckoosandbox.org/

Table 3: The number of instructions of the whole trace, virtualized snippets, and kernels in the Multi-VM experiment. S1 is the virtualized snippet by tiger-white and S2 is the virtualized snippet by fish-black. K1 and K2 are the kernels of S1 and S2 respectively.

Programs	T	S1	S2	S1+S2	K1	K2	K1+K2	(S1+S2)/T(%)	(K1+K2)/T(10^{-4})
grep	1,217,671	122,615	231,807	354,422	537	1,458	1,995	29.1	16.4
bzip2	1,594,486	120,103	206,049	326,152	713	1,540	2,253	20.8	14.1
aes	2,566,455	110,801	240,743	351,544	792	1,675	2,467	13.7	9.6
md5sum	2,310,301	138,508	249,138	387,646	649	1,549	2,198	16.8	9.5
thttpd	3,691,011	123,080	277,226	400,306	711	1,563	2,274	10.8	6.2
sqlite	4,764,819	143,995	294,373	438,368	802	1,898	2,700	9.2	5.7

Table 4: VMHunt evaluation result on malware samples. S/T is calculated in percentage. K/T is calculated in (10^{-4})

Name	Type	T	S	K	S/W	K/W
chodebot	Botnet	1,967,000	150,129	930	5.9	4.7
nzm	Botnet	6,141,556	181,457	1432	3.0	2.3
phatbot	Botnet	2,224,405	152,723	1008	6.9	4.5
zswarm	Botnet	5,587,140	168,529	1395	3.0	2.5
tsgh	Botnet	5,199,837	145,372	1362	3.0	2.6
dllinject	Virus	8,634,893	174,232	1568	2.0	1.8
locker_builder	Virus	10,435,886	198,695	1293	1.9	1.2
locker_locker	Virus	4,594,868	146,960	893	3.2	1.9
temp_java	Virus	8,661,836	185,380	1504	2.1	1.7
tears	Ransom	2,658,615	143,308	1074	5.4	4.0

is invoked. If the key generation procedure is cracked, people can easily calculate the key by themselves. So the key generation procedure is the sensitive area in the ransomware. The developer adopts virtualization to hide this procedure.

According to the final result of VMHunt's analysis, the key generation procedure can be represented as the following formula. The symbol t is a 32-bit integer representing the time when invoking the ransomware. k is the 128-bit key for AES encryption. k is generated by the concatenation of four encoding operations on t.

$$k = [t \oplus \texttt{0xabcd1234}, \texttt{shl}(t, 4), t \wedge \texttt{0xdeadbeef}, t + 1]$$

Particularly, based on our analysis, the virtualization tool translates the $t + 1$ operation to 10 instructions in the VM, which add and subtract constants as follows.

$$t + 1 = t - 522959822 - 4 + 20 + 8 - 4 - 16 + 4 + 522959846 - 522959866 + 522959835$$

We rebuild the control flow graph from the virtualized snippet and the kernel as shown in Appendix Figure 10. The kernel is significantly simple than the virtualized snippet. We also looked into the VM implementation and verified that the VM is implemented in threaded model. There is no explicit dispatch loop inside the execution trace. What's more, we find a fake dispatch loop iterating 872 times. Each time it fetches an integer from an array and jump to different fake handlers based on the integer value. The fake handlers are all junk code. Our kernel extraction effectively filter out all of those redundant sections.

8.5 Unvirtualized Programs

The effectiveness of VMHunt could be hurt by too many false positives. We are curious about whether VMHunt produces false positives on benign programs without virtualization. Therefore, we apply VMHunt to testbed programs with all virtualization options turned off. The result shows that the virtualization boundary detection does report several false positives. For example, some function calls happen to use all registers. Those false positives are filtered out in the following process when the virtualized kernel is extracted. In benign programs, the program snippet extracted is almost the same as the original trace since they are not obfuscated. Only the true virtualized kernel is significantly smaller than the original trace. Table 5 shows the size of the snippets extracted in the boundary detection and virtualized kernel detection in the experiment on grep. The virtualization boundary detection report 7 snippets as possible virtualization snippet, in which two of them (snippet 4 and 7) are real virtualization snippets. As shown in Table 5, in the snippets without virtualization, the kernel is the majority part of the snippet because there is only a few redundant code. In contrast, in the virtualized snippets, only very few instructions constitute the kernel because lots of instructions are redundant. Therefore, the ratio between kernel and the snippet is a good metric to filter out the false positives. In our experiment, we set 90% as the threshold to distinguish true virtualization snippet.

Besides, another observation is that the true virtualized snippet is significantly longer than the benign snippets, because the virtualized snippet includes the whole execution of a VM. This feature is also used for filtering out the false positives. In VMHunt, we set the threshold for snippet length as 10,000. These thresholds work perfectly in our experiment. As the final result, VMHunt accurately recognizes all virtualized snippets and reports zero false positive.

8.6 Performance

Overall, there are two phases in VMHunt, trace logging and offline analysis. The trace logging component is built upon Intel Pin [38], a dynamic binary instrumentation tool. The overhead of the trace logging is typically about 5X slow down. Table 6 show the execution time of every component in the offline phase. The boundary detection time increases as the trace size increases. The total time of analyzing one program is about 20 minutes. Since all testbed programs are real-world programs rather than synthetic examples, VMHunt's performance is good for practical virtualization analysis.

Table 5: The number of instructions of the snippets extracted in the boundary detection and kernel detection in the grep experiment. Snippet 4 and 7 are true virtualization snippets. S is the virtualized snippet size. K is the kernel size.

Snippet	S	K	K/S(%)
1	5,371	5,103	95.01
2	218	218	100.00
3	3,557	3,282	92.27
4	130,329	552	0.42
5	1,697	1,572	92.63
6	2,392	2,288	95.61
7	168,857	1061	0.63

Table 6: VMHunt's offline analysis performance. BD is boundary detection. K-Extraction is kernel extraction. MGSE is multiple granularity symbolic execution. The execution time is measured in minutes.

Programs	BD	K-Extraction	MGSE	Total
grep	7.2	4.8	5.3	17.3
bzip2	9.3	3.7	4.7	17.7
aes	10.9	4.1	6.3	21.3
md5sum	11.4	4.9	5.8	22.1
thttpd	14.7	4.7	5.1	24.5
sqlite	16.9	5.1	6.7	28.7

9 DISCUSSION

In this section, we discuss VMHunt's limitations, possible countermeasures, and future work. First, VMHunt bears with the same incompleteness as any dynamic analysis: every time only one execution path can be sufficiently analyzed. The possible mitigation is to automatically generate new inputs to explore uncovered paths through concolic execution [42] or guided fuzz testing [28]. Second, an attack to VMHunt's trace logging is to fingerprint dynamic binary instrumentation environment and then exit exceptionally [49]. We can strengthen VMHunt by running malware in a transparent environment [34, 84]. Our multiple granularity symbolic execution is effective to defeat data encodings via bitwise operations, which are quite common in commercial obfuscators. Attackers can mislead the detection of VM context switch by inserting redundant context switch instructions. We can defend this attack in two ways: 1) these redundant instructions can be removed in the simplification procedure; 2) we can check whether the switched context is actually used in the kernel; if not, the context switch instructions are considered to be redundant. Some work strengthens code obfuscation by diversifying VM contexts and handler functions [35, 75, 76]. However, VMHunt's semantics-based simplification is able to deal with the code mutation effects.

Theoretically, if the whole program is virtualized, VMHunt is hard to locate it because no context switch occur in this case. However, We wish to reiterate that whole program virtualization rarely happens in practice. First, existing virtualization technique cannot correctly handle some common program structures and instructions, so whole program virtualization will result in compatibility problems. Second, whole program virtualization translate the entire program to VM instructions and interpret them during runtime, which will cause significant slowdown.

In all of our tested VMs, context switch instructions save and restore the content for all general registers, because the VM execution uses all of them. It is possible to customize a VM that only uses some of the registers, so the context switch instructions would only save and restore those used registers. This design will affect our context switch instruction clustering. One possible solution is to check whether only those saved registers are used between the context switch instructions. In practice, we have not observed any virtualization obfuscator using partial registers. Using partial registers complicates the design of VM greatly. Especially, fewer number of available registers lead to more register spilling, which results in worse performance.

10 RELATED WORK

Deobfuscation of code virtualization. Code virtualization is one of the strongest obfuscation available to malware authors, and thus automatic deobfuscation methods can assist rapid understanding of malicious code. As the decode-dispatch based emulator is the classic, simple way to virtualize program code [63], a large portion of the previous works focus on reverse engineering this class of code virtualization [29, 30, 51, 55, 59]. A representative work, *Rotalumé* [59], uses dynamic analysis to detect the central decode-dispatch loop and then find the mappings between bytecode and related handler functions, whose control flow graphs are constructed for malware analysis. However, the latest commercial obfuscators have adopted two improvements to evade detection: 1) threaded interpretation in which the central decode-dispatch loop does not exist any more; 2) fake decode-dispatch loops to mislead loop search. In contrast, VMHunt is a generic approach that reveals better resilience to these evasions. Note that the obfuscated handler functions can be further optimized to better understand their semantics [9, 10, 22, 37, 57], and VMHunt's simplification approach is orthogonal to them.

The approaches in the second category do not require the assumption of emulator structure. Instead, they attempt to select the execution instructions that have control/data dependencies with original code semantics [19, 40, 67, 83]. For example, Kevin et al. perform equational reasoning [18] to identify the instructions that affect system call arguments. They treat such instructions as an approximation to the original code [19]. BinSim [40] achieves the similar results through an enhanced backward slicing. However, they may disregard the protected code that do not affect observable behaviors. Dynamic taint analysis is also applied to removing the instructions related to the dispatcher structure [67, 83], but only taint source dependent instructions can be kept. In comparison, VMHunt's slicing starts from multiple sources at VM boundary, making the resulting instructions more complete.

Another difference is that the correctness testing is seldom done by the previous deobfuscation work, while our simplification result is provable. We use a theorem prover to prove the simplified code is semantically equivalent to the original code. Some work only measured the similarity of control flow graphs (e.g., around

80% similarity) [59, 83] or x86 instruction opcode [19], but they didn't prove their deobfuscation result has the same behavior as the original program.

Symbolic execution of binary code. Symbolic execution has emerged as a fundamental technique for automatically analyzing binary code [3, 4, 72, 82]. Many laborious security analysis tasks, such as control flow de-obfuscation [7, 41], exploit generation [2, 6, 12], and cryptographic function detection [81], have been recast as a set of constraint satisfaction problems. Then advanced Satisfiability Modulo Theories (SMT) solvers are utilized to solve these constraints efficiently. To harness the full strength of SMT solvers, the key is to accurately abstract domain-specific security analysis task as verification constraints. VMHunt's multiple granularity symbolic execution reverts data encoding effects and produces concise constraints that could otherwise be hard to solve. VMHunt advances the use of symbolic execution in obfuscated binary code analysis.

11 CONCLUSION

Code virtualization is one of the most advanced software obfuscation techniques. Because of the high performance penalty and incomplete compatibility, code virtualization is mainly used to protect selected code segments. Existing virtualization deobfuscation work are either ad hoc, designed for a specific emulator structure, or assuming the scope of virtualized code is known to security analysts *a priori*. This paper presents a novel approach called *VMHunt*, a generic approach to locate virtualization-obfuscated code and simplify it. We consider the common virtual machine context switch behavior as a general detection feature, and optimize the obscure virtualized code through a semantics-based slicing and multiple granularity symbolic execution. Our evaluation shows that VMHunt can accurately identify the virtualized section and greatly simplify it by several orders of magnitude. Our study demonstrates VMHunt is an appealing complement to malware analysis.

ACKNOWLEDGMENTS

We thank the CCS anonymous reviewers and Heng Yin for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1652790, and the Office of Naval Research (ONR) grants N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894. Jiang Ming was also supported by the University of Texas System STARs Program.

REFERENCES

[1] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. 2014. Current Trends and the Future of Metamorphic Malware Detection. In *Proceedings of the 7th International Conference on Security of Information and Networks (SIN'14)*.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation on Source Code. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.

[3] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation Against Symbolic Execution Attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*.

[4] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*.

[5] Sebastian Banescu, Ciprian Lucaci, Benjamin Krämer, and Alexander Pretschner. 2016. VOT4CS: A Virtualization Obfuscation Tool for C#. In *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*.

[6] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[7] S. Bardin, R. David, and J. Y. Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[8] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (1973).

[9] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*.

[10] Ian Blumenfeld, Roberta Faux, and Paul Li. 2013. SMT Solvers for Malware Unpacking. In *Proceedings of the 11th InternationalWorkshop on Satisfiability Modulo Theories (SMT'13)*.

[11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*.

[12] D. Brumley, P. Poosankam, D. Song, and J. Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P'08)*.

[13] Joshua Cazalas, J. Todd McDonald, Todd R. Andel, and Natalia Stakhanova. 2014. Probing the Limits of Virtualized Software Protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW'14)*.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S^2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

[15] Christian Collberg. last reviewed, 10/01/2017. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/.

[16] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, Chapter 4.4, 258–276.

[17] C. Collberg, C. Thomborson, and D. Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'98)*.

[18] Kevin Coogan and Saumya Debray. 2011. Equational Reasoning on x86 Assembly Code. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*.

[19] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*.

[20] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)* (2016).

[21] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. 2003. The Case for Virtual Register Machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*.

[22] Fabrice Desclaux and Camille Mougey. 2017. Miasm: Reverse Engineering Framework. RECON.

[23] Anthony Desnos. 2010. Dynamic, Metamorphic (and opensource) Virtual Machines. Hack.lu.

[24] Robert B. K. Dewar. 1975. Indirect Threaded Code. *Commun. ACM* 18, 6 (1975).

[25] M. Anton Ertl and David Gregg. 2001. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In *Proceedings of the 2001 European Conference on Parallel Processing*.

[26] Nicolas Falliere, Patrick Fitzgerald, and Eric Chien. 2009. Inside the Jaws of Trojan.Clampi. Symantec Technical Report.

[27] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)*.

[28] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[29] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in Computer Virology* 6, 3 (2010).

[30] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17)*.

[31] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. 2013. API Chaser: Anti-analysis Resistant Malware Analyzer. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'13)*.

[32] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*.

[33] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)*.

[34] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security'14)*.

[35] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, Tianzhang Xing, Guixin Ye, Jie Zhang, and Zheng Wang. 2016. Exploiting Dynamic Scheduling for VM-Based Code Obfuscation. In *Proceedings of the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'16)*.

[36] Boris Lau. 2008. Dealing with Virtualization Obfuscators. CARO Workshop.

[37] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. 2017. Deobfuscation of Virtualization-obfuscated Code through Symbolic Execution and Compilation Optimization. In *Proceedings of the 19th International Conference on Information and Communications Security (ICICS'17)*.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*.

[39] Ramya Manikyam, J. Todd McDonald, William R. Mahoney, Todd R. Andel, and Samuel H. Russ. 2016. Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)*.

[40] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*.

[41] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[42] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (S&P'07)*.

[43] Philip OKane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* 9, 5 (2011).

[44] Oreans Technologies. 2015. Protecting Better with Code Virtualizer. http://oreans.com/codevirtualizer.php.

[45] Oreans Technologies. last reviewed, 10/01/2017. Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php.

[46] Oreans Technologies. last reviewed, 10/01/2017. Themida: Advanced Windows Software Protection System. https://www.oreans.com/themida.php.

[47] Joshua Phillips, Vitaly Zaytsev, and Abhishek Karnik. 2009. Parasitics: The Next Generation. Kaspersky Lab Technical Report.

[48] Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*.

[49] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17)*.

[50] Michalis Polychronakis. 2011. *Reverse Engineering of Malware Emulators*. Springer US, Chapter Encyclopedia of Cryptography and Security.

[51] Jason Raber. 2013. Virtual Deobfuscator: Removing virtualization obfuscations from malware. Black Hat USA.

[52] Ben Read and Jonathan Leathery. 2017. CVE-2017-0199 Used as Zero Day to Distribute FINSPY Espionage Malware and LATENTBOT Cyber Crime Malware. FireEye Threat Research Blog.

[53] ReWolf. last reviewed, 10/01/2017. x86 Virtualizer. http://www.openrce.org/blog/view/847/x86_Virtualizer_-_source_code.

[54] Thomas Roccia. 2017. Malware Packers Use Tricks to Avoid Analysis, Detection. McAfee Blogs.

[55] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT'09)*.

[56] Kevin A. Roundy and Barton P. Miller. 2013. Binary-code Obfuscations in Prevalent Packer Tools. *Comput. Surveys* 46, 1 (2013).

[57] Jonathan Salwan and Sébastien Bardin and Marie-Laure Potet. 2017. Deobfuscation of VM based software protection. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'17)*.

[58] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'15)*.

[59] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*.

[60] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. 2005. Virtual Machine Showdown: Stack Versus Registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*.

[61] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*.

[62] Craig Smith. 2008. Creating Code Obfuscation Virtual Machines. RECON.

[63] 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., San Francisco, CA, USA.

[64] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08). Keynote invited paper.*

[65] Aditya K Sood, Richard J Enbody, and Rohit Bansal. 2011. SpyEye malware infection framework. Virus Bulletin.

[66] StrongBit Technology. last reviewed, 10/01/2017. EXECryptor: Bulletproof software protection. http://www.strongbit.com/execryptor.asp.

[67] Zhanyong Tang, Lei Wang, Kaiyuan Kuang, Chao Xue, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, and Zheng Wang. 2017. SEEAD: A Semantic-based Approach for Automatic Binary Code De-obfuscation. In *Proceedings of 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'17)*.

[68] Clark Taylor and Christian Collberg. 2016. A Tool for Teaching Reverse Engineering. In *Proceedings of the 2016 USENIX Workshop on Advances in Security Education.*

[69] The Enigma Protector. last reviewed, 10/01/2017. Enigma Protector: A professional system for executable files licensing and protection. http://enigmaprotector.com/.

[70] Tora. 2012. Devirtualizing FinSpy. POC 2012.

[71] 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 the 36th IEEE Symposium on Security & Privacy (S&P'15)*.

[72] Julien Vanegue, Sean Heelan, and Rolf Rolles. 2012. SMT Solvers for Software Security. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT'12)*.

[73] VMProtect Software. last reviewed, 10/01/2017. VMProtect software protection. http://vmpsoft.com.

[74] C. Wang, J. Davidson, J. Hill, and J. Knight. 2001. Protection of software-based survivability mechanisms. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*.

[75] Huaijun Wang, Dingyi Fang, Guanghui Li, Na An, Xiaojiang Chen, and Yuanxiang Gu. 2014. TDVMP: Improved Virtual Machine-Based Software Protection with Time Diversity. In *Proceedings of the 3rd Program Protection and Reverse Engineering Workshop.*

[76] Huaijun Wang, Dingyi Fang, Guanghui Li, Xiaoyan Yin, Bo Zhang, and Yuanxiang Gu. 2013. NISLVMP: Improved Virtual Machine-Based Software Protection. In *Proceedings of the 9th International Conference on Computational Intelligence and Security.*

[77] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. 2016. Translingual Obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P'16)*.

[78] Zhenxiang Jim Wang. 2010. Virtual Machine Protection Technology and AV industry. CARO Workshop.

[79] Josh Watson. 2017. An extra bit of analysis for cLEMENCy. Trail of Bits Blog.

[80] Haijiang Xie, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Nightingale: Translating Embedded VM Code in x86 Binary Executables. In *Proceedings of the 20th Information Security Conference(ISC'17)*.

[81] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[82] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[83] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*.

[84] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*.

[85] Qinghua Zhang and Douglas S. Reeves. 2007. MetaAware: Identifying Metamorphic Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*.

[86] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Proceedings of the 8th International Workshop on Information Security Applications (WISA'07)*.

Appendix A

A common virtualization procedure is shown as follows.
a) In the source code of the program that will be virtualized, insert marks around the sensitive area. In C/C++ code, typically the mark is implemented as macros. Figure 8 shows such an example.
b) Compile the source code using a normal compiler like GCC or MS VC++. The code is linked to a library provided by the virtualization tool. The result is an executable file. In this step, the sensitive area in the executable file is not obfuscated. It is only marked.
c) Run the virtualization tool to process the executable file. The virtualization tool will translate the marked area to the virtualized code and append the virtual machine in the binary code.

```
1  ...
2  #include "VirtualizerSDK.h"
3
4  int f(int a)
5  {
6      int b = 1;
7
8      VIRTUALIZER_START   // The macro marks the
9                          // starting point of the
10                         // virtualized area
11     a++;
12
13     VIRTUALIZER_END     // The macro marks the ending
14                         // point of the virtualized
15                         // the area
16
17     return a + b;
18 }
```

Figure 8: An example showing the virtualization marks in the source code to be virtualized. The macros mark the starting and ending point of the sensitive area which will be virtualized.

A.0.1 Sensitive Area. According to the manual book of the virtualization tools, users should prevent the following cases when applying virtualization to a program.

- Users should avoid virtualizing a loop that repeats many times to avoid too much performance loss.
- Switch/Case statements and exception handling inside a sensitive area might not work properly after virtualization. Therefore, those program structures are not recommended to be virtualized.

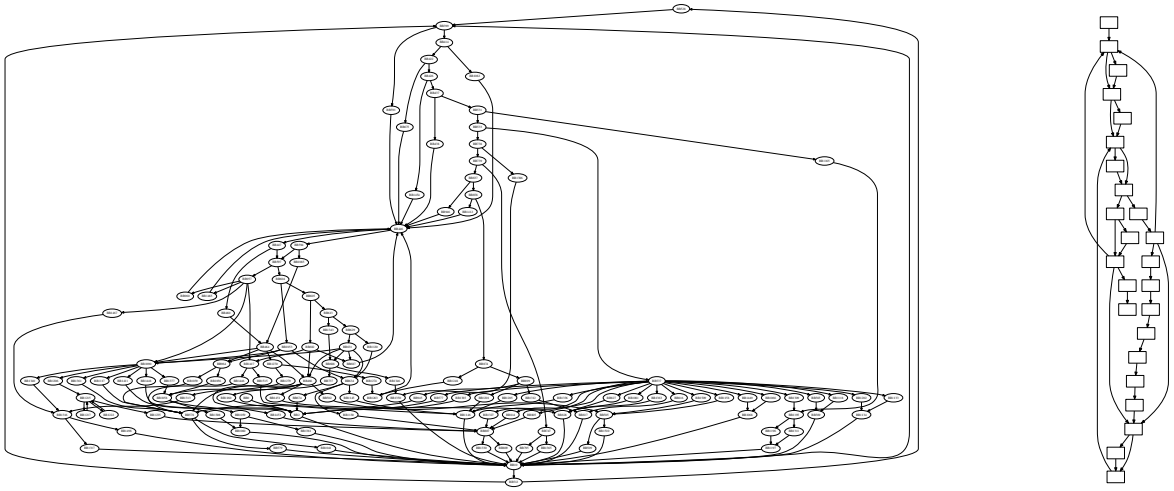
In practice, due to the performance overhead and the compatibility problems, virtualization can only be applied to limited program structures and areas. The recommended way of applying virtualization is only protect the sensitive area in your program.

One typical situation of the sensitive area is the checking procedure in a trial/registration scheme of an application. The same application can run in two modes, the trial mode or the registered mode. The registered mode provides full features whereas the trial mode only provide limited features. Figure 9 shows an example of the registration checking function and the recommended way of protecting them by virtualization. The function `f` runs different branches based on the value of the global variable `reg_mode`. Therefore, all snippets that reads or write `reg_mode` should be considered as sensitive areas and should be virtualized.

Basically, the trial/registration scheme needs a checking mechanism to decide in which mode the application should run. For example, the checking mechanism can be implemented as a function to verify whether a serial number is eligible. Figure 9 shows an example of the registration checking function and the recommended way of protecting them by virtualization. The global boolean variable at the first line stores the mode of the application. `true` means the program is running under the registered mode and `false` means the trial mode. At line 8 in the main function, the application check whether it is registered by calling the function `checkRegistration()`. The function will return `true` if the serial number is eligible or `false` if not. The function `f` includes two branches for the registered mode and trial mode, respectively. It checks the global variable `reg_mode` and then select one branch to execute. In this example, the global variable `reg_mode` is sensitive because the trial/registration scheme can be work around if we can modify its value. Therefore, any snippet that reads or write it should be considered as sensitive area and should be virtualized.

```
1  bool reg_mode;
2
3  int main ()
4  {
5      ...
6
7      VIRTUALIZER_START
8      reg_mode = checkRegistration(); // sensitive area 1
9      VIRTUALIZER_END
10
11     ...
12 }
13
14 void f()
15 {
16     VIRTUALIZER_START
17     if (reg_mode) {               // sensitive area 2
18         // code for registered verion
19     } else {
20         // code for trial version
21     }
22     VIRTUALIZER_END
23 }
```

Figure 9: An example showing a registration checking function and the sensitive area being virtualized.



(a) Virtualized snippet.

(b) Kernel.

Figure 10: The CFG recovered from the virtualized snippet and kernel.