Handling Anti-Virtual Machine Techniques in Malicious Software

HAO SHI, USC/Information Sciences Institute
JELENA MIRKOVIC, USC/Information Sciences Institute
ABDULLA ALWABEL, USC/Information Sciences Institute

Malware analysis relies heavily on the use of virtual machines for functionality and safety. There are subtle differences in operation between virtual and physical machines. Contemporary malware checks for these differences and changes its behavior when it detects VM presence. These anti-VM techniques hinder malware analysis. Existing research approaches to uncover differences between VMs and physical machines use randomized testing, and thus cannot guarantee completeness.

In this paper we propose a detect-and-hide approach, which systematically addresses anti-VM techniques in malware. First, we propose *cardinal pill testing* – a modification of red pill testing that aims to enumerate the differences between a given VM and a physical machine, through carefully designed tests. Cardinal pill testing finds five times more pills by running fifteen times fewer tests than red pill testing. We examine the causes of pills and find that, while the majority of them stem from the failure of VMs to follow CPU specifications, a small number stem from under-specification of certain instructions by the Intel manual. This leads to divergent implementations in different CPU and VM architectures. Cardinal pill testing successfully enumerates the differences that stem from the first cause. Finally, we propose *VM Cloak* – a WinDbg plug-in, which hides the presence of virtual machines from malware. VM Cloak monitors each executed malware command, detects potential pills, and modifies at run time the command's outcomes to match those that a physical machine would generate. We implemented VM Cloak and verified that it successfully hides VM presence from malware.

CCS Concepts: •**Security and privacy** → **Malware and its mitigation; Software reverse engineering;**

Additional Key Words and Phrases: System security, virtual machine testing, reverse engineering, assembly

ACM Reference Format:
Hao Shi, Jelena Mirkovic, and Abdulla Alwabel, 2016. Handling Anti-Virtual Machine Techniques in Malicious Software. *ACM Trans. Priv. Secur.* 0, 0, Article 0 (0000), 30 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Today's malware analysis [2; 3; 4; 5; 6] relies on virtual machines to facilitate fine-grained dissection of malware functionalities (e.g., Anubis [7], TEMU [9], and Bochs [10]). For example, virtual machines can be used for taint analysis, OS-level information retrieval, and in-depth behavioral analysis. Use of VMs also protects the host through isolating it from malware's destructive actions.

Malware authors have devised a variety of evasive behaviors to hinder automated and manual analysis of their code, such as anti-dumping, anti-debugging, anti-virtualization, and anti-intercepting [11; 12]. Kirat et al. [13] detect 5,835 malware samples (out of 110,005) that exhibit evasive behaviors. The studies in [14; 15] show

This material is based upon work supported by the Department of Homeland Security, and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-10-C-2018.
Author's addresses: H. Shi, J. Mirkovic, and Abdulla Alwabel, Information Sciences Institute, University of Southern California.

DOI: 0000001.0000001

that anti-virtualization and anti-debugging techniques have become the most popular methods of evading malware analysis. Chen et al. [16], find in 2008 that 2.7% and 39.9% of 6,222 malware samples exhibit anti-virtualization and anti-debugging behaviors respectively. In 2011, Lindorfer et al. [15] detect evasion behavior in 25.6% of 1,686 malicious binaries. In 2012, Branco et al. [14] analyze 4 million samples and observe that 81.4% of them employ anti-virtualization and 43.21% employ anti-debugging.

Upon detection of a virtual environment or the presence of debuggers, malicious code can alternate execution paths to appear benign, exit programs, crash systems, or even escape virtual machines. Therefore, it is critically important to devise methods that handle anti-virtualization and anti-debugging, to support future malware analysis. In this paper, we focus only on anti-virtualization handling.

We observe that malware can differentiate between a physical and a virtual machine due to numerous subtle differences that arise from their implementations. Let us call the physical machine an *Oracle*. Malware samples can execute sets of instructions with carefully chosen inputs (aka *pills*), and compare their outputs with the outputs that would be observed in an Oracle. Any difference leads to detection of VM presence. In addition to these *semantic attacks*, there are two other approaches to anti-virtualization – timing and string attacks (see Section 2). Our work focuses heavily on detecting and handling semantic attacks as they are the most complex. Our solution, however, also handles timing and string attacks.

Semantic attacks are successful because there are many differences between VMs and physical machines, and existing research in VM detection [1; 17; 18] uses randomized tests that cannot fully enumerate these differences. We observe that when a malware is run within a VM, all its actions are visible to the VM and all the responses are within a VM's control. If differences between a physical machine and a VM could be enumerated, the VM or the debugger could use this knowledge to provide expected behaviors when malware commands are executed, thus hiding VM presence. This is akin to kernel rootkit functionality, where the rootkit hides its presence by intercepting instructions that seek to examine processes, files and network activity, and provides replies that an uncompromised system would produce.

In this paper, we propose cardinal pill testing [19], an approach that attempts to enumerate all the differences between a physical machine and a virtual machine *that stem from their differences in instruction execution*. These differences can be used for CPU semantic attacks (see Section 2). Our contributions include the following:

(1) We improve on the previously proposed red pill testing [1; 17] by devising tests that carefully traverse operand space and explore execution paths in instructions with the minimal set of test cases. We use 15 times fewer tests and discover 5 times more pills than red pill testing. Our testing is also more efficient: 47.6% of our test cases yield a pill, compared to only 0.6% of red pill tests. In total, we discover between 7,487 and 9,255 pills depending on the virtualization technology and the physical machine being tested.
(2) We find two root causes of pills: (1) failure of virtual machines to strictly adhere to CPU design specification and (2) vagueness of the CPU design specification that leads to different implementations in physical machines. Only 2% of our pills stem from the second phenomenon.

We originally propose cardinal pill testing in [19]. In this paper we make the following additional contributions:

(1) We evaluate kernel-space instructions to cover the whole Intel x86 instruction set (publication [19] only evaluated user-space instructions). In our evaluation, kernel-space instructions show a much higher yield rate (pills/test cases) than user-space

ones: 83.5%∼85.5% v.s. 38.5%∼47.7%, depending on different virtualization modes of VMs.
(2) We propose VM Cloak – a WinDbg plug-in, which hides VM presence. VM Cloak monitors malware execution of each instruction, and modifies malware states after the execution if the instruction matches a pill. The modified states match the states that would be produced if the instruction were executed on a physical machine.
(3) We implement VM Cloak and evaluate it through two data sets. We first randomly select and analyze 319 malware samples captured in the wild, to evaluate how frequently are anti-VM techniques used by contemporary malware. Then we perform closer evaluation using three known samples that have been demonstrated to show heavy anti-VM behavior. We show that malware, run under VM Cloak and within a VM, exhibits the same file and network activities as malware run on a bare metal machine. This proves that VM Cloak successfully hides the VM from malware.
(4) We implement handling of timing and string attacks, while our prior work [19] only handled semantic attacks.

All the scripts and test cases used in our study will be publicly released at our project website (https://steel.isi.edu/Projects/cardinal/).

2. ANTI-VIRTUALIZATION TECHNIQUES

Anti-virtualization techniques can be classified into three broad categories [16; 18]:

Semantic Attacks. Malware targets certain CPU instructions that have different effects when executed under virtual and real hardware. For instance, the `cpuid` instruction in Intel IA-32 architecture returns the `tsc` bit with value `0` under the Ether [21] hypervisor, but outputs `1` in a physical machine [22]. As another example found in our experiment, when moving hex value `7fffffffh` to floating point register `mm1`, the resulting `st1` register is correctly populated as `SNaN` (signaling non-number) in a physical machine, but has a random number in a QEMU-virtualized machine. Malware executes these pills and checks their output to identify the presence of a VM.

Timing Attacks. Malware measures the time needed to run an instruction sequence, assuming that an operation takes a larger amount of time in a virtual machine compared to a physical machine [12]. Contemporary virtualization technologies (dynamic translation [23], bytecode interpretation [10], and hardware assistance [21]) all add significant delays to instruction execution [1].

String Attacks. VMs leave a variety of traces inside guest systems that can be used to detect their presence. For instance, QEMU assigns the "QEMU Virtual CPU" string to the emulated CPU and similar aliases to other virtualized devices such as hard drive and CD-ROM. A simple query to Windows registry will reveal the VM's presence [16].

The main focus of our work is on handling semantic attacks as they are the most complex category to explore and enumerate. The string attacks can be handled through enumeration and hiding of VM traces, which can be done by comprehensive listing and comparison of files, processes, and Windows registries, with and without virtualization. Also, timing attacks can be handled through systematic lying about the VM clock. Our VM Cloak system implements detection and hiding of all three classes of attacks, but our intellectual contributions focus mostly on semantic attacks.

3. RELATED WORK

In this section we discuss the work related to handling of semantic attacks (pill testing and pill hiding) as well as handling of other anti-virtualization techniques.

[1]This method can also be used to detect debuggers, because stepping code adds large delays.

3.1. Pill Testing

Martignoni et al. present the initial red pill work in EmuFuzzer [1]. They propose red pill testing – a method that performs a random exploration of a CPU instruction set and parameter spaces, to look for pills. Testing is performed by iterating through the following steps: (1) initialize input parameters in the guest VM, (2) duplicate the content in user-mode registers and process memory in the host, (3) execute a test case, (4) compare resulting states of register contents, memory and exceptions raised—if there are any differences, the test case is a pill. In their follow-up work KEmuFuzzer [17], the authors extend the state definition to include the kernel space memory, and test cases are embedded in the kernel to facilitate testing of privileged instructions. However, the authors test boundary and random values for explicit input parameters but do not examine implicit parameters, while we attempt to evaluate implicit parameters as well.

In their recent work [20], they use symbolic execution to translate code of a high-fidelity emulator (Bochs) and then generate test cases that can investigate all discovered code paths. Those test cases are used to test a lower-fidelity emulator such as QEMU. While this symbolic analysis can automatically detect the differences between a high-fidelity and a low-fidelity model, it is difficult to evaluate how accurately their high-fidelity model resembles a physical machine. In addition, the authors exclude test generation for floating-point instructions since their symbolic execution engine does not support them. In our work, we use instruction semantics to carefully craft test cases that explore all code paths. We also use bare-metal physical machines as Oracle, which improves fidelity of tests and helps us discover more pills.

Other works [24; 15; 25] focus on detecting anti-virtualization functions of malware based on profiling and comparing their behavior in virtual and physical machines. They do not uncover the details of anti-virtualization methods that each individual binary employs, and they can only detect anti-virtualization checks deployed by their malware samples, while we detect many more differences that could be exploited in future anti-virtualization checks.

3.2. Pill Hiding

Dinaburg et al. [21] aim to build a transparent malware analyzer, Ether, by implementing analysis functionalities out of the guest using Intel VT-x extensions for hardware-assisted virtualization. However, nEther [22] finds that Ether still has significant differences in instruction handling when compared to physical machines, and thus anti-VM attacks are still possible, i.e., Ether does not achieve complete transparency.

Kang et al. [18] propose an automated technique to dynamically modify the execution of a whole-system emulator to fool a malware sample's anti-emulation checks. They first collect two execution traces of a malware sample: one reference trace that the authors believe passes all its anti-VM checks and contains real, malicious behavior, and the other trace in which the sample fails certain anti-VM checks. For example, a physical machine or a high-fidelity VM can be used to generate the reference trace, and a low-fidelity VM produces a trace that shows anti-VM behavior. Then, the authors use a trace matching algorithm to locate the point where emulated execution diverges. Finally, they compare the states of the reference system and the VM to create a dynamic state modification that repairs the differences. But these VM modifications are specific to a particular malware sample, while our work handles anti-VM attacks in a universal way, across different malware samples.

Other works specify a variety of anti-VM techniques but they do not propose a systematic framework to detect and handle all the attacks. For example, Ferrie [12] shows some attacks against VMware, VirtualPC, Bochs, QEMU, and other VM prod-

ucts. While the attacks are effective in detecting the VMs, no methodology is illustrated to protect the VMs from being detected.

3.3. Timing and String Attacks

Timing Attack. To feed malware with the correct time information, Vasudevan et al. [**?**] replace `rdtsc` instruction with a `mov` instruction that stores the value of their internal processor-counter to the `eax` register. However, it is unclear how they maintain the internal processor counter. In addition, malware can query a variety of time sources besides using `rdtsc` to fetch the time-stamp counter. The authors of [**?**] apply a clock patch, thereby resetting the time-stamp counter to a value that mimics the latency close to that of normal execution. This work claims that it also performs the same reset on the real-time clocks since malware could use the real-time clock. Nevertheless, the details of clock resetting are unclear, and the enumeration of different time sources are not provided.

String Attacks. Chen et al. [16] propose that malware may mark a system as "suspicious", if they find that certain tools are installed with well-known names and in a well-known location, such as "VMWare" and "OllyDbg". However, they do not provide a systematic method to hide the presence of these strings. Vasudevan et al. [**?**] merely state that they overwrite memory data that leaks the presence of debuggers with values copied from physical machines. The details of memory data and the overwriting method are not mentioned.

3.4. Towards a Transparent Malware Analysis Framework

In addition to the above anti-VM techniques, malware authors have also devised a variety of other anti-analysis attacks such as anti-debugging. For example, malware may remove the breakpoints set by a debugger, disable keyboard input, or obfuscate its disassembly code rendered. Therefore, researchers have been seeking a transparent malware analysis framework that can minimize its exposure to malware. Unfortunately, all of the current methodologies have been proven to be detectable by malware. We will classify these frameworks based on the high-level concepts behind them, and illustrate how malware detects them in Section 7.

4. CARDINAL PILL TESTING

In this section, we first introduce our testing infrastructure that enables the evaluation of the same test cases on different pairs of virtual and physical machines. Then, we discuss the fundamental intuition behind our test case generation model, independent of Instruction Set Architecture (ISA). Finally, we apply our generation model on Intel x86 instruction set and describe how we group them to automate test case generation as best as we can.

4.1. Testing Architecture

Our testing architecture consists of three physical machines: a master, a slave hosting a virtual machine (VM), and a slave running a bare-metal as a reference (Oracle). The slaves are connected to the master by serial wires. The master generates test cases (Section 4.2) and schedules their execution in slaves. In both slaves, we configure a daemon that helps the master set up a specific test case in each testing round.

The execution logic of our cardinal pill testing is illustrated in Figure 1. The master maintains a debugger that issues commands to and transfers data back from the slaves. The Oracle and the VM have the same test case set and the same daemon; we only show one pair of test case and daemon in Figure 1 for clarity. We set the slaves in the kernel debugging mode so that they can be completely frozen when necessary.

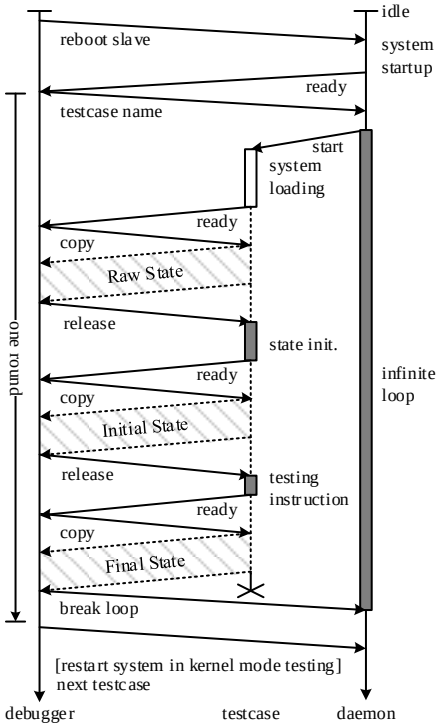


Fig. 1. Logic Execution

At the beginning, the master reboots the slave (either VM or Oracle) for fresh system states. After the slave is online, the daemon signals its readiness to the master, which then evaluates test cases one per round.

We define the *state* of a physical or virtual machine as a set of all user and kernel registers, and the data stored in the part of code, data, and stack segments, which our test case accesses for reading or writing. In addition, the state also includes any potential exceptions that may be thrown.

During each round, the master interacts with the slave through three main phases. In the first phase, it issues a test case name to the daemon that resides in a slave, then the daemon will ask the slave system to load this test case stored in its local disk. Afterwards, the system starts allocating memory, handles, and other resources needed by the test case program. When this *system loading* completes, the test case executes an interrupt instruction (`int 3`), which notifies the master and halts the slave. At this point, the master saves the *raw state* of the slave locally. We use this raw state to identify *axiom pills* (see Section 4.2) instead of discarding it [1; 17].

In the second phase, the master releases the slave to execute the test case's initialization code and raise the second interrupt. Instead of using the same initial system state for all test cases, we carefully tailor register and memory for each test case, such that all possible exceptions and semantic branches can be evaluated (Section 4.2). The master copies the resulting *initial state* and releases the slave again.

In the third phase, the slave executes the actual instruction being tested and raises the last interrupt. The master will store this *final state* and use it to determine whether the tested instruction along with the initial state is a cardinal pill (see Section 5.1). A test case may drive the slave into an infinite loop or crash itself or its OS. To detect

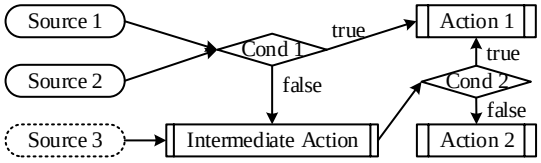


Fig. 2. Defined Behavioral Model of Instruction

this, we set up an execution time limit for each test case, so that the master can detect incapacitated slaves and restore them.

Finally, when evaluating test cases with user-space commands, we can set up the next test case after the previous one has completed. After evaluation of test cases with kernel-space commands, and after evaluation of test cases that crash the OS, we must reboot the system before proceeding with testing.

4.2. Behavioral Model of Instruction

In a modern computer architecture, the program's instructions are usually executed in a pipeline style: fetching instructions from memory, decoding registers and memory locations used in the instruction, and executing the instruction. This pipeline can be modeled as a directed, multi-source, and multi-destination graph, as shown in Figure 2. Each source node stands for the input parameters that are demanded by the instruction. They may be explicitly required by the instruction (solid line) in its mnemonic or implicitly needed by its specification (dashed line). These parameters will be examined by certain condition checks and may go through some intermediate processing (Intermediate Action). Finally, the execution of the instruction may end up with different operations (Action 1 or Action 2) depending on the intermediate checks and actions. At parameter fetching and action, exceptions may occur due to a variety of causes. For example, the memory location of a source may be inaccessible (memory page not present or address out of range). The intermediate action may cause an overflow, which will throw an overflow exception. Furthermore, the purpose of the instruction itself may be to raise an exception.

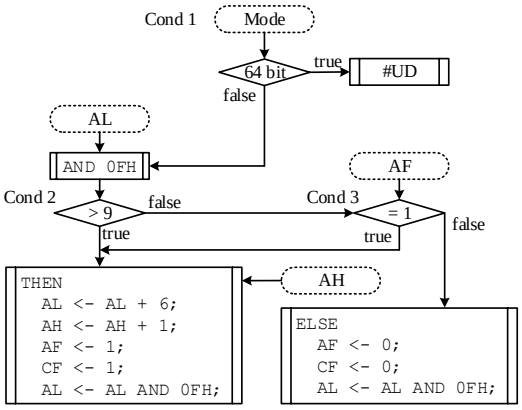
In most cases, the behavioral model of an instruction does not specify how certain registers will be updated, because they are not consumed or produced by the instruction. This incomplete specification leaves room for different implementations by different vendors. We found in our evaluation that these registers may still be modified by CPU. We call these modifications *undefined behaviors*. Because we do not know the logic behind the undefined behaviors, there is no sound methodology to completely evaluate them, other than exhaustive search. But exhaustive search is impractical because the space of instruction parameters is prohibitively large. We briefly discuss our attempt to infer semantics of undefined behaviors and thus reduce the need for exhaustive search in Section 5.3.

The goal of VMs is to faithfully virtualize the behavioral model for each instruction of the ISA that they are emulating, including both normal and abnormal execution paths. Based on these observations, we set up the following goals of our test case generation algorithm:

— For defined behaviors of a given instruction, all execution branches should be evaluated. All flag bit states that are read explicitly or implicitly, or updated using results must be considered.
— All potential exceptions must be raised, such as memory access and invalid input arguments.

```
IF 64-Bit Mode
  THEN
    #UD;
  ELSE
    IF ((AL AND 0FH)>9)
        or (AF=1))
      THEN
        AL <- AL + 6;
        AH <- AH + 1;
        AF <- 1;
        CF <- 1;
        AL <- AL AND 0FH;
      ELSE
        AF <- 0;
        CF <- 0;
        AL <- AL AND 0FH;
    FI;
FI;
```

(a) Specification



(b) Defined Behavioral Model

Fig. 3. Building Defined Behavioral Model for aaa Instruction

— Undefined behaviors should be investigated to reveal undocumented implementation specifics.

In the following section, we illustrate how we generate test cases based on the defined behavioral models of Intel x86 instructions.

4.3. Generating Test Cases for Intel x86 Instruction Set

Intel x86 instruction set is one of the most complex ISAs, which contains about 1000 instructions. Most of them incorporate multiple execution paths. We first illustrate our test case generation approach on an example instruction – aaa – and then describe our general approach.

4.3.1. An Example – aaa. The aaa instruction adjusts the sum of two unpacked binary coded decimal (BCD) values to create an unpacked BCD result. Its specification from Intel manual is shown in Figure 3(a), and the corresponding behavioral model is illustrated in Figure 3(b). This instruction has no explicit parameters but needs four implicit parameters: system mode, and the al, the ah, and the af registers. The al and the ah are 8-bit registers and the af is a one-bit flag in eflags register. The behavioral model contains one intermediate action (and 0fh) and three sink actions (#ud, then, and else nodes).

We aim to generate a minimal set of test cases that explore all possible code paths in this instruction's defined behavioral model. In addition, we want to cover all the boundary values that are used in the condition checks. Therefore, we will generate the following test cases as shown in Table I. In the first test case, we set the mode to be 64 bit, so an #ud exception will be thrown. The second test sets both al and ah registers to contain the minimal value, and sets af to 0. This test case evaluates the ELSE action at the end. The following test cases populate al, ah, and af to evaluate all possible execution paths and boundary values used in the instruction's defined behavioral model.

4.3.2. Test Case Template under Windows x86 Platform. We use Windows x86 platform for demonstration purposes because this is the most popular OS that has been targeted by malware. After we derive the minimal test case set that explores all execution paths of an instruction's behavioral model, the next step is to map the set to concrete binaries

Table I. Test Cases Generated for aaa's Defined Behavioral Model

No.	Mode/Cond 1	AL/Cond 2	AF/Cond 3	AH	Testing Goals
1	64 bit/true	N/A	N/A	N/A	bound(Cond 1), #ud exception
2	32 bit/false	0/false	0/false	0	min(AL), min(AH), and ELSE
3	32 bit/false	0/false	0/false	0ffh	min(AL), max(AH), and ELSE
4	32 bit/false	0/false	1/true	0c9h	min(AL), rand(AH), and THEN
5	32 bit/false	9/false	0/false	58h	bound(Cond 2), rand(AH), and ELSE
6	32 bit/false	9/false	1/true	0a6h	bound(Cond 2), rand(AH), THEN
7	32 bit/false	0ffh/true	0/false	0	max(AL), min(AH), and THEN
8	32 bit/false	0ffh/true	0/false	30h	max(AL), rand(AH), and THEN
9	32 bit/false	0ffh/true	0/false	0ffh	max(AL), max(AH), and THEN
10	32 bit/false	0ffh/true	1/true	0b3h	max(AL), rand(AH), and THEN
11	32 bit/false	8/false	0/false	8ah	bound(Cond 2), rand(AH), and THEN
12	32 bit/false	10/true	1/true	3fh	bound(Cond 2), rand(AH), and THEN
13	32 bit/false	3/false	0/false	07fh	rand(AL), rand(AH), and ELSE

that can be executed. In order to achieve this goal, we first compose a test case template to describe the initialization work that is the same for all test cases, as shown in Figure 4. This program notifies the master in Figure 1 and then halts the slave as soon as it enters the main function (line 2), so the master can save the states. The same interaction happens at lines 27, 29, and 38, after the test case completes a certain step. Then the program installs a structured exception handler for the Windows system (line 4 – 7). If an exception occurs, the program will jump directly to line 31, so we can save the system state before exception handling.

From line 9 to 25, we perform *general-purpose initialization*. Registers and memory are populated using pre-defined values, including all floating point and integer formats. This step occurs in all test cases and the carefully chosen, frequently used values are stored in the registers to minimize the need for specific initialization. Afterwards, the *specific initialization* (line 26) makes tailored modifications to the numbers if needed for a given test case. For example, the eax is set to 1bh at line 10 for all test cases. One particular test case may need 0ffh value in this register and will update it at line 26. The actual instruction is being tested at line 28, where all defined and undefined behaviors will be evaluated in various test cases.

Now we describe an example of mapping the second test case of aaa in Table I to our test case template. The placeholder [state_init] at line 26 will be replaced by the four instructions shown in the upper block in Figure 4. The sahf instruction transfers bits 0-7 of ah into the eflags register, which correctly sets af to 0. Since aaa does not take any explicit parameters, [testing_insn] at line 28 will become aaa in all test cases for this instruction. When compiling test cases, we disable linker optimization and use a fixed base address. This eases the interaction between the master and slaves, and does not affect the testing outcome. In our testing, we find that physical machines also set or reset the sf, zf, and pf flags. These flags are not defined for the aaa instruction in the manual, hence this is the undefined behavior of aaa.

4.3.3. Extending to Intel x86 Instruction Set. In this section, we describe how to apply our test case generation method to the entire Intel x86 instruction set. We manually analyze instruction execution flows defined in Intel manuals [26], group the instructions into semantically identical classes, and classify all possible input parameter values into ranges that lead to distinct execution flows. We then draw random parameter values from each range.

The IA-32 CPU architecture contains about 1000 instruction codes. In our test design strategy, a human must reason about each code to identify its inputs and outputs and how to populate them to test all execution behaviors. To reduce the scale of

```
 1 main proc
 2   int 3               ; Raw State
 3
 4   push offset handler ; install SEH
 5   assume fs:nothing
 6   push fs:[0]
 7   mov  fs:[0], esp
 8
 9   ;; populate reg and memory
10   mov eax, 0000001bh
11   mov ebx, 00001000h
12   ...
13   ;; double precision floating-point
14   mov eax, 00403080h
15   mov dword ptr [eax], 0h
16   mov dword ptr [eax+4], 7ff00000h ; +Infi
17   ...
18   ;; single precision floating-point
19   mov eax, 0040318ch
20   mov dword ptr [eax], 0ff801234h  ; SNaN
21   ...
22   ;; double-extended precision FP
23   ...
24   ;; unsupported double-extended precision
25   ...
26   [state_init]      ; specific init
27   int 3             ; Initial State
28   [testing_insn]    ; instruction in test
29   int 3             ; Final State
30   call ExitProcess
31 handler:
32   ;; push exception information onto stack
33   mov edx, [esp + 4]        ; excep_record
34   mov ebx, [esp + 0ch]      ; context
35   push dword ptr [edx]      ; excep_code
36   ...
37   push dword ptr [edx + 0c0h] ; eflags
38   int 3           ; Final State (exception)
39   mov eax, 1h
40   call ExitProcess
41 main endp
42 end main
```

Example initialization for `aaa`:

```
mov ah, 46h
sahf          ; set AF to 0
mov al, 0     ; populate AL
mov ah, 0     ; populate AH
```

Example testing for `aaa`:

```
aaa
```

Fig. 4. Test Case Template (in MASM assembly)

this human-centric operation, we first group the instructions into six categories: arithmetic, data movement, logic, flow control, miscellaneous, and kernel. The arithmetic and logic categories are subdivided into general-purpose and FPU categories based on the type of their operands. We then define parameter ranges to test per category, and adjust them to fit finer instruction semantics as described below. This grouping greatly reduces human time investment and reduces the chances of human errors. It took one person from our team two months to devise all test cases. Table II shows the number of different mnemonics, examples, and parameter ranges we evaluate for each category.

Arithmetic Group. We classify instructions in this group into two subgroups, depending on whether they work solely on integer registers (general-purpose group), or on floating point registers (FPU group) as well. The instructions in the FPU group include instructions with x87 FPU, MMX, SSE, and other extensions.

Table II. Instruction Grouping

Category	Insn. Count	Example Instructions	Parameter Coverage
arithmetic	48	`aaa`, `add`, `imul`, `shl`, `sub`	min, max, boundary values, randoms in different ranges
	336	`addpd`, `vminss`, `fmul`, `fsqrt`, `roundpd`	±infi, ±normal, ±denormal, ±0, SNaN, QNaN, QNaN floating-point indefinite, randoms
data mov	232	`cmova`, `fild`, `in`, `pushad`, `vmaskmovps`	valid/invalid address, condition flags, different input ranges
logic	64	`and`, `bound`, `cmp`, `test`, `xor`	min, max, boundary values, $>$, $=$, $<$, flag bits
	128	`andpd`, `vcomiss`, `pmaxsb`, `por`, `xorps`	±infi, ±normal, ±denormal, ±0, SNaN, QNaN, QNaN FP indefinite, $>$, $=$, $<$, flag bits
flow ctrl	64	`call`, `enter`, `jbe`, `loopne`, `rep stos`	valid/invalid destination, condition flags, privileges
misc	34	`clflush`, `cpuid`, `mwait`, `pause`, `ud2`	analyze manually and devise dedicated input
kernel	52	`arpl`, `int`, `lds`, `lgdt`, `ltr`, `wbindbv`	devise parameter values covering all input ranges and boundaries if applicable

Based on the argument types and sizes, branch conditions, and the number of arguments, we divide both subgroups into finer partitions. For example, `aaa`, `aas`, `daa`, and `das` in the general-purpose subgroup all compare the `al` register (holding one packed BCD argument 8-bits long) with `0fh` and check the adjustment flag `af` in the `eflags` register. This decides the output of the instruction. To test instructions in this set, we initialize the `al` register to minimal (`00h`), maximal (`0ffh`), boundary (`0fh`), and random values in different ranges (`[01h, 0eh]`, `[10h, 0feh]`). We also set `af` to `0` and `1` for different `al` values.

If a mnemonic takes two parameters, we select at least three value pairs to ensure that a greater-than, equal-to, and less-than relationship between them is satisfied in our test set. For the FPU subgroup, the parameter ranges are separated based on the sign, biased exponent, and significand, which splits all possible values into 10 domains: ±infi, ±normal, ±denormal, 0, SNaN, QNaN, and QNaN floating-point indefinite. We sample values from all these ranges to test behaviors in the arithmetic FPU group. For example, `fadd`, `fsub`, `fmul`, and `fdiv` each use one operand that can be specified using four different addressing modes; one of them is `m64fp`, which stands for a double precision float stored in memory. These instructions add/sub/mul/div the `st(0)` register with the operand's value and store the result in `st(0)`. In addition, they also read control bits in the `mxcsr` register and `fdiv` checks the divide-by-zero exception. In our test cases, we generate values for the two floating point operands from the 10 identified ranges and permute the relevant bits in the `mxcsr` register. Because instructions in this subgroup can also access memory to read operands, we devise additional test cases to evaluate the memory management unit. We place the `m64fp` argument in and out of the valid address space of a data segment, into a segment with and a segment without required privileges, and into a segment that is paged in and a segment that is paged out of memory. By combining these test cases together, all potential memory access exceptions can be raised along with all potential arithmetic exceptions.

Data Movement. Data Movement instructions have simple behavioral semantics. They either move data between registers, between co-processors, or between registers and memory. While moving data is straightforward, the exception behavior can be more complex and nuanced. Therefore, we focus our parameter coverage on values

that would cause exceptions such as invalid addresses, segments, and register states. For example, an exception will be raised if we move data from the FPU stack to the general purpose registers when the FPU stack is empty. All the input parameters and the states that influence an instruction's execution outcome must be tested. Similarly, there are 30 conditional move instructions, and we also ensure that the condition flags states are fully explored during testing.

Logic Group. Logic instructions test relationship and properties of operands and set flag registers correspondingly. We divide these instructions into general-purpose and FPU depending on whether they use `eflags` register only (general-purpose) or they use both `eflags` and `mxcsr` registers (FPU). We also partition this group based on the flag bits they read and argument types and sizes. When designing test cases, in addition to testing min, max, and boundary values for each parameter, for instructions that compare two parameters, we also generate test cases where these parameters satisfy larger-than, equal-to, and less-than conditions.

For example, one of the subgroups has `bt`, `btc`, `btr`, and `bts` instructions because all of them select a bit from the first operand at the bit-position designated by the second operand, and store the value of the bit in the carry flag. The only difference is how they change the selected bit: `btc` complements; `btr` clears it to `0`; and `bts` sets it to `1`. The first argument in this subgroup of instructions may be a register or a memory address of size 16, 32, or 64, and the second must be a register or an immediate number of the same size. If the operand size is 16, for example, we generate four input combinations (choosing the first and the second argument from `0h, 0ffffh` values), and we repeat this for `cf = 0` and `cf = 1`. Furthermore, we produce three random number combinations that satisfy less-than, equal-to, and greater-than relationships. While the operand relationship does not influence execution in this case, it does for other subgroups, e.g., the one containing `cmp`.

In the FPU subgroup, we apply similar rules to generate floating point operands. We generate test cases to populate the `mxcsr` register, which has control, mask, and status flags. The control bits specify how to control underflow conditions and how to round the results of SIMD floating-point instructions. The mask bits control the generation of exceptions such as the denormal operation and invalid operation. We use `ldmxcsr` to load `mxcsr` and test instruction behaviors under these scenarios.

Flow Control. Similar to logic instructions, flow control instructions also test condition codes. Upon satisfying jump conditions, test cases start execution from another place. For short or near jumps, test cases do not need to switch the program context; but for far jumps, they must switch stacks, segments, and check privilege requirements.

The largest subgroup in this category is the conditional jump `jcc`, which accounts for 53% of flow control instructions. Instructions in this group check the state of one or more of the status flags in the `eflags` register (`cf`, `of`, `pf`, `sf`, and `zf`) and if the required condition is satisfied, they perform a jump to the target instruction specified by the destination operand. A condition code (`cc`) is associated with each instruction to indicate the condition being tested for. In our test cases, we vary the status flags and set the relative destination addresses to the minimal and maximal offset sizes of byte, word, or double word as designated by mnemonic formats. For example, `ja rel8` jumps to a short relative destination specified by `rel8` if `cf = 0` and `zf = 0`. We permute `cf` and `zf` values in our tests, and generate the destination address by choosing boundary and random values from the ranges `[0, 7fh]` and `[8fh, 0ffh]`.

For far jumps like `jmp ptr16:16`, the destination may be a conforming or nonconforming code segment or a call gate. There are several exceptions that can occur. If the code segment being accessed is not present, a `#NP` (not present) exception will

be thrown. If the segment selector index is outside descriptor table limits, an exception #GP (general protection) will signal the invalid operand. We devise both valid and invalid destination addresses to raise all these exceptions in our test cases.

Miscellaneous. Instructions in this group provide unique functionalities and we manually devise test cases for each of them that evaluate all defined and undefined behaviors, and raise all exceptions.

Kernel instructions. Kernel instructions are supposed to run under ring 0 and each of them accomplishes specific tasks. For example, arpl adjusts the rpl of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. The int instruction raises a numbered interrupt, and ltr loads the source operand into the segment selector field of the task register. For this category, we devise parameter values that can cover all input ranges and boundaries where applicable.

5. DETECTED PILLS

We use two physical machines in our tests as Oracles: (**O1**) an Intel Xeon E3-1245 V2 3.40GHz CPU, 2 GB memory, with Windows 7 Pro x86, and (**O2**) Xeon W3520 2.6GHz, 512MB memory, with Windows XP x86 SP3. The VM host has the same hardware and guest system as the first Oracle, but it has 16 GB memory, and runs Ubuntu 12.04 x64. We test QEMU (VT-x and TCG), and Bochs, which are the most popular virtual machines deploying different virtualization technologies: hardware-assisted, dynamic translation, and interpretation respectively. We allocate to them the same size memory as in the Oracle. We test QEMU versions 0.14.0-rc2 (**Q1**, used by EmuFuzzer), 1.3.1 (**Q2**), 1.6.2 (**Q3**), and 1.7.0 (**Q4**), and Bochs version 2.6.2. The master has an Intel i7 CPU and installs WinDbg 6.12 to interact with the slaves. For test case compilation, we use MASM 10 and turn off all optimization. Our user-space test cases take around 10 seconds to run on a physical machine and 15 – 30 seconds to run on a VM. The kernel-space test cases need about 5 minutes per case, because they need a system reboot.

Counting different addressing modes, there are 1,769 instructions defined in Intel manual [26]. Out of these, there are 958 unique mnemonics. Following our test generation strategy (Section 4.2), we generate 19,412 and 593 test cases for user-space and kernel-space instructions respectively.

5.1. Evaluation Process

We classify system states into user registers, exception registers, kernel registers, and user memory. The user registers contain general registers such as eax and esi. The exception registers are eip, esp, and ebp. The differences in the exception registers imply differences in the exceptions being raised. The kernel registers are used by the system and include gdtr, idtr, and others. In our evaluation of user-space test cases, we do not populate kernel registers in the initialization step because this may crash the system or lead it to an unstable status. We simply use the default values for kernel registers after system reboot. The contents of kernel registers are saved as part of our states and compared to detect differences between physical and virtual machines.

For each test case, we first examine whether the user registers, exception registers, and memory are the same in the Oracle and the VM in the initial state. If they are different, it means that the VM fails to virtualize the initialization instructions (line 26 in Figure 4) to match their implementation in the Oracle. We mark this test case as "fatal". If the initial values in these locations agree with each other, we then compare the final states. A test case will be tagged as a pill when user registers, kernel registers, exception registers, or memory in the final states are different.

Table III. Results Overview

VMs	Pills	Crash	Fatal
User-space testing (19,412)			
Q1 (TCG)	9,255/47.7%	7/<0.1%	1,378/7%
Q2 (TCG)	9,201/47.4%	7/<0.1%	1,376/7.1%
Q1 (VT-x)	7,523/38.7%	2/<0.1%	3/<0.1%
Q2 (VT-x)	7,478/38.5%	2/<0.1%	0/0%
Bochs	8,958/46.1%	2/<0.1%	950/4.9%
Kernel-space testing (593)			
Q1 (TCG)	495/83.5%	14/2%	3/0.5%
Q2 (TCG)	496/83.6%	14/2%	3/0.5%
Q1 (VT-x)	506/85.3%	2/0.3%	3/0.5%
Q2 (VT-x)	506/85.3%	2/0.3%	3/0.5%
Bochs	507/85.5%	2/0.3%	3/0.5%

5.2. Results

In this section, we discuss our evaluation results from multiple perspectives. (1) We show the detailed spectrum of the pills found by our test cases. (2) By comparing our pills with those found by [1], our approach discovers much more pills using much fewer test cases. (3) We investigate the root causes of our pills and find four more categories that are not captured by [1; 20]. (4) In order to identify the pills that are persistent across distinct physical and virtual machines, we generate 2,915 additional test cases for 13 selected instructions (Section 5.2.4). (5) Finally, we discuss axiom pills that are found during the raw states upon system loading.

5.2.1. Pill Spectrum. Table III shows the results of testing several virtual machines against Oracle1 (O1). The second column "pills" shows the number of pills for different VMs. Both QEMU (TCG) and Bochs exhibit moderate transparency – almost half of the test cases report different states between O1 and VMs. For Q2 (VT-x) 38.5% of user-space test cases result in pills, but there were no fatal cases. The pills we find for Q2 (VT-x) occur because QEMU does not preserve the fidelity provided by hardware. Therefore, we should be careful when using hardware-assisted VMs for fidelity purposes. Their transparency depends on how they utilize the hardware extension.

The third column "crash" counts test cases that crash the system. For QEMU (TCG), one test case crashes the Oracle 1 and another one crashes the virtual machine. Another five crash both of them. For QEMU (VT-x) and Bochs, two test cases crash both the physical and the virtual machine. The number of fatal test cases are shown in the last column. All of them are related to FPU movement instructions. In some test cases that use denormals, SNaN, or QNaN values, the virtual machines could not populate the operand register as required. We note that no fatal test cases are found for VT-x technology.

For kernel-space test cases, we observe that the yield rates of the pills are much higher than those of user-space test cases. A closer investigation reveals that most of pills are due to differences in exceptions. We believe the higher yield rates of kernel-space test cases are due to the fact that the setting up of exception in kernel mode requires more work, which is more error-prone for VMs.

Table IV shows the breakdown of pills per instruction category for user-space test cases in Table II. The FPU arithmetic, FPU logic and data movement categories contain the most pills—around 83%. Table V shows the breakdown of the pills with regard to the resource that is different between a physical and a virtual machine in the final state. Most pills occur due to differences in the kernel registers.

5.2.2. Comparison with EmuFuzzer Pills. EmuFuzzer [1] generates 3 million user-space test cases and the authors randomly select 10% of the cases to test in different virtual

Table IV. Pills per Instruction Category (User-space)

Category		Q1 (TCG)	Q2 (TCG)	Q1 (VT-x)	Q2 (VT-x)	Bochs	Total tests
arithmatic	general	877	872	633	626	920	2,702
	FPU	4,525	4,486	3,619	3,603	4,245	6,743
data movement		1,788	1,780	1,539	1,524	1,804	4,394
logic	general	371	365	345	346	363	2,185
	FPU	1,446	1,447	1,132	1,127	1,362	2,192
flow control		164	166	172	169	171	1,017
miscellaneous		84	85	83	83	93	179
total		9,255	9,201	7,523	7,478	8,958	19,412

Table V. Details of pills with regard to the resource being different in the final state—in some cases multiple resources will differ so the same pill may appear in different rows

Category	Q2 (TCG)	Q2 (VT-x)	Bochs
user register	2,416	34	1,671
excp register	1,578	21	1,566
kerl register	8,398	7,457	8,572
data content	46	9	20

machines. The authors publish 20,113 red pills for QEMU0.14.0-rc2, which is about 7% of the tested cases. Because they do not publish the entire test case set, we cannot directly compare our test cases with theirs. Instead we compare their yield with ours (percentage of tests cases that result in a pill), and the total number of unique pills found. We also verify if we found all 20,113 pills that were published by EmuFuzzer researchers.

Out of our 19,412 test cases we find 9,255 pills, which is 47.6% yield, while EmuFuzzer's yield is 1,850/300,000 = 0.6%. Higher yield means shorter testing time.

To compare the number of pills found, we define a *unique pill* as a pill whose mnemonic and parameter value combination does not appear in any other pill. We use the same QEMU version as EmuFuzzer (Q1 (TCG)) and run all the 20,113 red pills they found. We successfully extract operand values for 20,102 pills. But among those there are only 1,850 unique red pills (9%) involving 136 different instruction mnemonics. Our 9,255 pills for Q1 (TCG) are all unique and involve 630 different instruction mnemonics. Thus we find five times more pills than Emufuzzer running $300,000/19,412 = 15$ times fewer tests, and cover 494 more mnemonics. We verify that we find all the 1,850 unique pills published by EmuFuzzer. Therefore, we conclude that our approach is more comprehensive and far more efficient than EmuFuzzer.

5.2.3. Root Causes of Pills. The differences detected by a pill can be due to registers, memory or exceptions that an instruction was supposed to modify, according to the Intel manual [26]. We call these instruction targets *defined resources*. However, there are a number of instructions defined in the Intel manual that may write to some registers (or to select flags) but the semantics of these writes are not defined by the manual. We say that these instructions affect *undefined resources*. For instance, the `aas` instruction should set the `af` and `cf` flags to `1` if there is a decimal borrow; otherwise, they should be cleared to `0`. The `of`, `sf`, `zf`, and `pf` flags are listed as affected by the instruction but their values are undefined in the manual. Thus, the `af` and `cf` flags are defined resources for the instruction `aas`, but `of`, `sf`, `zf`, and `pf` flags are undefined.

Table VI shows the number of pills that result from differences in undefined and defined resources for each instruction category compared to Oracle 1. We note that a small number of pills that relate to general-purpose arithmetic and logic instructions

Table VI. Pills using Undefined/Defined Resources

Category		Q2 (TCG)	Q2 (VT-x)	Bochs
arith	gen	195/677	0/626	194/726
	FPU	0/4,486	0/3,603	0/4,245
data mov		0/1,780	0/1,524	0/1804
logic	gen	23/342	0/346	20/343
	FPU	0/1,447	0/1,127	0/1,362
flow ctrl		0/166	0/169	0/171
misc		0/85	0/83	0/93
kernel insn.		0/496	0/506	0/507

occur because of different handling of undefined resources by physical and virtual machines. These comprise roughly 2% of all the pills we found.

For pills originating from defined resources in both user and kernel space, we analyze their root causes and compare them against those found by the symbolic execution method [20]. We find all root causes listed in [20] that are related to general-purpose instructions and QEMU's memory management unit.

Because the symbolic execution engine in [20] does not support FPU instructions, we discover additional root causes that are not captured by the symbolic execution method. First, we find that QEMU does not correctly update 6 flags and 8 masks in the `mxcsr` register when no exception happens, including invalid operation flag, denormal flag, precision mask, overflow mask. It also fails to update 7 flags in `fpsw` status register such as stack fault, error summary status, and FPU busy. Second, QEMU fails to throw five types of exceptions when it should, which are: float_multiple_traps, float_multiple_faults, access_violation, invalid_lock_sequence, and privileged_instruction. Third, QEMU tags FPU registers differently from Oracles. For example, it sets `fptw` tag word to "zero" when it should be "empty", and sets it to "special" when "zero" is observed in Oracles. Finally, the floating-point instruction pointer (`fpip`, `fpipsel`) and the data pointer (`fpdp`, `fpdpsel`) are not set correctly in certain scenarios.

5.2.4. Identifying Persistent Pills. Differences found in our tests between an Oracle and a virtual machine may not be present if we used a different Oracle or a different virtual machine, i.e. a difference may stem more from an implementation bug specific to that CPU or VM version than from an implementation difference that persists across versions. Furthermore, outdated CPUs may not support all instruction set extensions that are available in recent ones. Finally, recent releases of VM software usually fix certain bugs and add new features, which may both create new differences and remove the old differences between this VM and physical machines. We hypothesize that *transient* pills are not useful to malware authors because they cannot predict under which hardware or under which virtual machine their program will run, and we assume that they would like to avoid false positives and false negatives.

To find pills that persist across hardware and VM changes, we perform our testing on multiple hardware and VM platforms. We select 13 general instructions that can be executed in all x86 platforms (`aaa`, `aad`, `aas`, `bsf`, `bsr`, `bt`, `btc`, `btr`, `bts`, `imul`, `mul`, `shld`, `shrd`) and generate 2,915 test cases for them to capture more pills that are caused by modification of undefined resources. We evaluate this set on the two physical machines (Oracle 1 and Oracle 2), three different QEMU versions (Q2, Q3, and Q4), and Bochs. We find 260 test cases that result in different values in `eflags` register in Oracle 1 and Oracle 2 and will thus lead to transient pills. Bochs' behavior for these test cases is identical to the behavior of Oracle 2. Out of the remaining 2,655 test cases, we find 989 persistent pills that generate different results in the three QEMU virtual machines when compared to the physical machines. They are all related to undefined resources.

Table VII. Undefined `eflags` Behaviors

Instruction	OF	SF	ZF	AF	PF	CF
aaa	0	0	ZF (ax)		PF (al + 6) or PF (al)	0
	0	0	ZF (al)		PF (al)	0
aad	F			F		F
	0			0		0
aam	0			0		0
aas	0	0	ZF (ax)		PF (al + 6 or al)	0
	0	0	ZF (al)		PF (al)	0
and, or, xor, text				0		
bsf, bsr	I	I		I	I	I
	0	0		F	0	0
bt, bts, btr, btc	I	I		I	I	
daa, das	0					
div, idiv	I	I	I	I	I	I
mul, imul		I	I	I	I	
		F	F	0	F	
		F	0	0	F	
rcl, rcr, rol, ror	I					
	F					
	OF(1-bit rotation)					
sal, sar, shl, shr shld, shrd	I			I		
	R			0		
	0			F		

Bochs performs surprisingly well and does not have a single pill for these particular test cases. Thus, we could not find persistent pills that would detect a VM for any given VM/physical machine pair in our tests, but we found pills that can differentiate between any of the QEMU VM versions and configurations that we tested, and any of the physical machines we tested.

We further investigate the persistence of pills that are caused by modifications to undefined resources, across different physical platforms. We select five physical machines with different CPU models in DeterLab [27]. Out of 218 pills that were found for Oracle 1 and Q2 (TCG), we were able to map 212 pills to all five physical machines (others involved instructions that did not exist in some of our CPU architectures). Fifty of those were persistent pills—the undefined resources were set to the same values in physical machines. We conclude that modifications to undefined resources can lead to pills that are not only numerous but also persistent in both physical and virtual machines. This further illustrates the need to understand the semantics of these modifications as this would help enumerate the pills and devise hiding rules for them without exhaustive tests.

5.2.5. Axiom Pills. In addition to comparing final states across different platforms we also compare raw states upon system loading. We define an *axiom* pill as a register or memory value whose raw state is consistently different between a physical machine and a given virtual machine. This pill can be used to accurately diagnose the presence of the given virtual machine. We select 15% of our test cases and evaluate them on Oracle 2, Q2, Q3 and Bochs. The axiom pills are shown in Table VIII. For example, the value of `0ffffffffh` in the `edx` register can be used to diagnose the presence of Q2 (VT-x).

5.3. Exploring Undefined Behavior Model

Our test cases were designed to explore effects of input parameters on defined resources. We thus claim that our test cases cover all specified execution branches for all instructions defined in Intel manuals. Our test pills should thus include all possible individual pills that can be detected for defined resources.

Table VIII. Axiom Pills

Reg	O1	Q1 (TCG)	Q2 (TCG)	Q1 (VT-x)	Q2 (VT-x)	Bochs
edx	vary	vary	vary	0ffffffffh	0ffffffffh	vary
dr6	0ffff0ff0h	0	0	0ffff0ff0h	0ffff0ff0h	0ffff0ff0h
dr7	400h	0	0	400h	400h	400h
cr0	8001003bh	8001003bh	8001003bh	8001003bh	8001003bh	0e001003bh
cr4	406f9h	6f8h	6f8h	6f8h	6f8h	6f9h
gdtr	vary	80b95000h	80b95000h	80b95000h	80b95000h	80b95000h
idtr	vary	80b95400h	80b95400h	80b95400h	80b95400h	80b95400h

We now explore the pills stemming from modifications to undefined resources, to evaluate their impact on the completeness of our pill sets and to attempt to devise semantics of these modifications. In our evaluation, we find that pills arising from undefined sources are due to the flags in `eflags`.

We analyze the instructions that affect one or more flags in the `eflags` register in an undefined manner. We generate additional test cases for each instruction to explore the semantics of modifications to undefined resources in each CPU. Although the exact semantics differ across CPU models, we consider four semantics of flag modifications that are the superset of behaviors we observed across tested hardware and software machines: a flag might be (1) cleared, (2) remain intact, (3) set according to the ALU output at the end of an instruction's execution, or (4) set according to an ALU output of an intermediate operation.

We run our test cases on a physical or virtual machine in the following manner. For each instruction, we set an undefined flag and execute an operation that yields a result inconsistent with the flag being set; for example, `zf` is set while the result is `0`. If the flag remains set we conclude that the instruction does not modify it. Similarly, we can test if the flag is set according to the final result. If none of these tests yield a positive result, we go through the sub-operations in a given instruction's implementation as defined in the CPU manual, and discover which one modifies the flag. For example: `aaa` adds `6` to `al` and `1` to `ah` if the last four bits are greater than `9` or if `af` is set. The instruction affects `of`, `sf`, `zf` and `pf` in an undefined manner. We find that in some machines, `zf` and `pf` are set according to the final result, while in others, `pf` is set according to an intermediate operation which is `al = al + 6`.

Table VII shows different semantics for each instruction, which are consistent across 5 different CPU models. Empty cells represent defined resources for a given instruction. Character "I" means the flag value is intact while "F" means that the flag is set according to the final result. Otherwise, the flag is set to the value in the cell.

To detect pills between a given virtual machine and one or many physical machines, we repeat the same tests on the virtual machine and look for differences in instruction execution semantics. If many physical machines are compared to a virtual machine, we look for such differences where physical machines consistently handle a given instruction in a way that is different from how it is handled in a virtual machine. For example, in Table VII, instruction `aad` either clears `of`, `af` and `cf` flags or sets them according to the final result. If a virtual machine were to leave these flags intact, we could use this behavior as a pill.

Our test methodology will discover all test pills (and thus all possible individual pills) related to modifications of undefined resources by user-space instructions *for a given physical/virtual machine pair*. Since the semantics of undefined resource modifications vary greatly between physical CPU architectures as well as between various virtual machines and their versions, all possible test pills cannot be discovered in a general case.

To summarize, our testing reveals pills that stem from instruction modifications to user-space or kernel-space registers. These modifications can further occur on defined

Table IX. Example Hiding Rule Generation for aaa Instruction

aaa	Before Execution			After Execution			
Parameter	AL	AH	eflags	AL	AH	eflags	Exception
Q1 (TCG)	0ffh	30h	246h	5	32h	257h	None
O1	0ffh	30h	246h	5	32h	217h	None

or on undefined resources for a given instruction. We claim we detect all test pills (and thus all the individual pills) that relate to modifications of defined resources. We can claim that because we fully understand semantics of these modifications, and all physical machines we tested strictly adhere to the instructions' semantics as specified in the manual. We cannot claim completeness for pills that relate to modifications of undefined resources because physical machine behaviors differ widely for those.

6. HANDLING ANTI-VM ATTACKS USING CARDINAL PILLS

In this section, we discuss how our cardinal pills can be utilized to improve the transparency of virtual machines.

6.1. Generating Hiding Rules from Cardinal Pills

While our cardinal pills are specific to Intel x86 architecture, they are not specific to any OS, VM, or debugger. Most malware analysis frameworks can use our pills to detect anti-VM attacks launched by malware.

We devise hiding rules from cardinal pills by transforming each pill into a {pre-condition, action} tuple. The pre-condition part is generated from the parameters. The action part defines how to change the VM's state to hide its presence, and it includes writing and reading of registers or memory locations and raising or suppression of exceptions. We provide more details below.

A malware analysis platform monitors each instruction and its parameters, and checks them against all pre-conditions. If there is a match with the pre-condition of a hiding rule, the platform implements the action specified in the rule.

Our action will, in many cases, lead to overwriting of the destination locations of the instruction (register or memory) with the values learned from a physical machine. For example, the No. 8 test case in Table I turns out to be a cardinal pill, as shown in Table IX. We first extract the condition of the pill as: `insn = aaa`, `al = 0ffh`, `ah = 30h`, and `eflags = 246h`. Then the hiding rule based on `O1` physical machine will be: `al = 5`, `ah = 32h`, and `eflags = 217h`. Other cardinal pills for `aaa` will be parsed in a similar way, and the combined set will be the final hiding rules for this instruction. Some instructions may lead to different exceptions being raised in a VM and a physical machine. For example, an instruction may not throw an exception in a VM but may throw it in an Oracle, or an instruction may throw different exceptions in these two platforms. In this case, we must throw the correct exception in a VM to match the Oracle. Our action for such instructions includes raising or suppression of the exceptions to match the exception state of a physical machine.

Some kernel instructions will retrieve values that are guaranteed to be different in VMs than in physical machines. For example, `cpuid` returns processor identification and feature information according to the input value entered initially in the `eax` register. Virtual machines usually do not provide the detailed information as a physical machine does. Therefore, malware can use this instruction to detect VMs. To handle this attack, we devise different values in `eax` to explore all execution paths of `cpuid`. When malware exploits this instruction to detect VMs, we return the values learned from a physical machine.

6.2. Integrating Hiding Rules with Existing Frameworks

Our hiding rules can be easily integrated with existing frameworks. For example, the infrastructure proposed in [18] may use our cardinal pills to improve the coverage of its application scenarios. In this work, the authors first collect one execution trace of a malware sample from a high-fidelity system (Ether) for reference, and then collect another trace from a low-fidelity system (QEMU). They believe the anti-VM checks of the sample fail in the reference system but succeed in the low-fidelity environment. Therefore, there are certain *diverging points* where two traces show different behaviors. Then the authors devise a Dynamic State Modification (DSM) infrastructure for the low-fidelity system to repair the differences automatically. This is achieved by changing the malware sample's observation of its environment to match the observations it makes on the reference system.

However, the DSM infrastructure is specific to a particular malware sample: the modification method cannot be applied to other samples using the same anti-VM checks. This limit can be improved by utilizing our cardinal pills as follows. During the active execution of a malware sample, DSM can monitor each instruction that the sample has executed. If the instruction together with its parameters match a cardinal pill, DSM will overwrite the VM state with the values observed in the physical machines. This way, the DSM module can be applied to more than one malware sample.

6.3. Integrating Hiding Rules with Debuggers – VM Cloak

Unfortunately, the DSM infrastructure is not available to public, so we could not use it with our cardinal pills. In this section, we describe how to use our cardinal pills in a debugger (WinDbg) to hide VM's presence from malware. Debuggers are widely used in malware analysis today. Malware may also detect debuggers using anti-debugging attacks and we discuss this limitation in Section 7.

We design VM Cloak as a WinDbg plug-in, which operates in single-stepping mode to avoid anti-disassembly behaviors in malware, such as packing and code overwriting. The execution logic of VM Cloak is shown in Figure 5. At the beginning, we utilize the disassembling function of WindDbg to disassemble the instruction following the current instruction pointer. Then, we match the disassembled instruction against criteria in our hiding rules, including CPU semantic attack, timing attack, and string attack (Section 2). If there is a match, we modify program states after executing this instruction, to hide VMs from malware. Now, we detail how VM Cloak can be used to handle different anti-VM techniques.

Semantic/Cardinal Pill Attacks. For this category of attack, VM Cloak tries to match each instruction against the correction rules. Upon a successful match, VM Cloak retrieves the expected behavior and reenacts it, overwriting registers and memory where needed.

Timing Attacks. VM Cloak maintains a software time counter and if it detects an instruction that reads system time, it returns a value using this time counter. We update our time counter by adding a small delta for each malware instruction that has been executed, and we make the delta's value vary with the complexity of the instruction. We also add a small, random offset to the final value of the time counter before returning the value to the application. This serves to defeat attempts to detect VM Cloak by running the same code twice and detecting exactly the same passage of time. VM Cloak maintains a list of instructions and system APIs that can be exploited by malware to query the time information, such as `rdtsc` and `GetTickCount()`. Whenever malware uses these methods, VM Cloak will replace the returned values with expected ones.

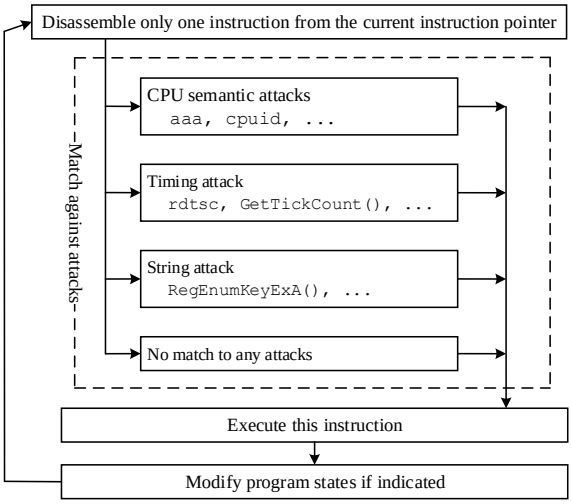


Fig. 5. Execution Logic of VM Cloak

Malware may use a sophisticated method to fetch the time information that we did not foresee in our research. For example, processor manufacturers may release new instructions to fetch time information or provide additional timing registers in the future. When a new attack mechanism is discovered by us or other researchers, VM Cloak can integrate its detection and handling easily. Finally, we currently cannot handle the cases when malware queries external time sources. This remains an open research problem [28].

String Attacks. VM Cloak monitors each instruction for use of APIs that query registry and files. If the values being read match a list of known VM-revealing strings (such as “vmware”, “vbox”, and “qemu”), we overwrite these strings with values from physical machines. We admit that it is hard to guarantee the completeness of this list as well as the list we maintain for timing attacks, because newly released VM products may introduce fresh strings and upcoming OS versions may provide new APIs for time queries. However, it is easy to extend VM Cloak to cover these new detection signals when they become available.

6.4. Evaluation

In this section, we evaluate VM Cloak using two data sets: (1) unknown samples captured in the wild, to measure prevalence of anti-VM approaches and (2) known malware samples that employ heavy anti-VM techniques, analyzed and published by other researchers, to test if VM Cloak can defeat these known techniques.

6.4.1. Evaluation Methodology. We evaluate each malware sample in two environments. First, we evaluate it within VM Cloak and under a VM. Second, we evaluate it on a physical machine, without any VM or debugger. We say that a VM is successfully hidden from malware if malware exhibits the same behavior in both environments. The scope of our evaluation is necessarily limited because: (1) there is no ground truth about which anti-VM checks are possible; (2) there is no well-understood representation of malware behavior, and thus we have to define our own.

We define malware behavior as a union of file and network activities. While malware may exhibit other behaviors, such as running calculations, invoking other applications, etc., we regard file and network activities as crucial for malware to export any knowl-

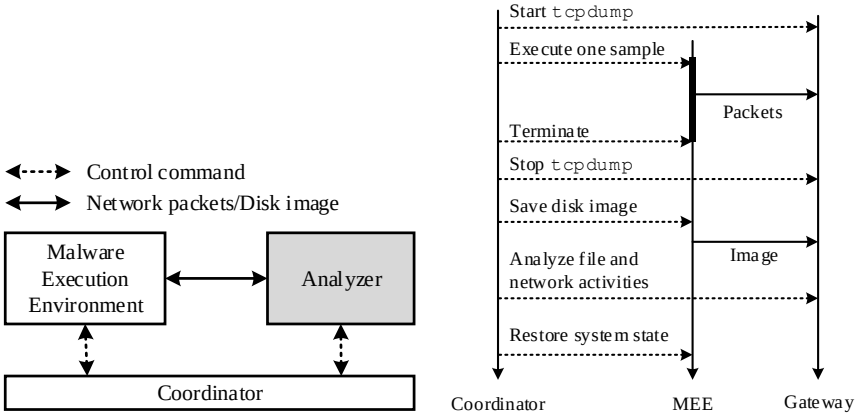
edge to an external destination or to receive external input (e.g., from a bot master). Thus, if a sample exhibits the same pattern of file and network activities within a VM (hidden by VM Cloak) and in a native run, we will conclude that we were able to successfully hide VM presence from malware. For file activities, we record file creations, deletions and modifications. We save the hard drive into raw disk images before and after running malware, and extract the file information using the SleuthKit framework [29; 13]. By comparing the file's meta data, we obtain the list of created, deleted, and modified files for each malware sample. Because malware may easily modify file path we do not use file path to infer malware behaviors. Instead, we use the file's contents. For network activities, we record the destination IPs, ports, and content of the traffic on our network interface, but we do not actually route the packets, to preserve the Internet from any harm.

Malware may have some randomness in its behavior, making it exhibit different activities over different runs. Operating system itself may trigger some activities, unrelated to malware's execution. To filter out noise, we first observe a base OS's file and network activities, without malware, in six runs. We create a union of all the created, deleted, and modified files, and all network communications – U_{base}. Next, we perform three native malware runs and create an intersection of activities found in all three runs – I_{native}. We define the set difference $S_{sig} = I_{native} - U_{base}$ as a malware's *signature* behavior. In our evaluation, we look for all the items from this signature to determine if malware performs the same malicious activities with and without VM Cloak.

6.4.2. Implementation Details. To analyze malware in an automatic way, we devise a testing environment as shown in Figure 6. There are three entities in Figure 6(a). Malware samples will be executed in the Malware Execution Environment (MEE), and their network traffic will be routed to Analyzer. The Coordinator is responsible for testing samples automatically, as shown in Figure 6(b). At the beginning, Coordinator will start the `tcpdump` service on Analyzer and then execute one sample on MEE. Each sample is automatically analyzed under VM Cloak for a maximum of 20 minutes. Next, Coordinator will save the disk image of MEE on Analyzer, using `dd` command. After the image is transmitted to Analyzer, Coordinator will load the disk of MEE with a fresh copy of the operating system. At the same time, Analyzer processes the network trace and disk image to extract the network and file activities of malware. Finally, another testing cycle will be launched. We run this testing environment on the DeterLab testbed. Each machine in our experiment has a 3GHz Intel processor, 2GB of RAM, one 36Gb disk, and 5 1-Gbps network interface cards.

6.4.3. Anti-VM is Popular: Testing with Samples from the Wild. We randomly select 527 malware binaries from Open Malware [**?**] that are captured in 2016. These samples are then sent to a malware analysis website VirusTotal [**?**], which uses approximately 50 anti-virus products to analyze each binary. We retain those binaries that are labeled as malicious by more than 50% anti-virus products, and this leaves us with 319 samples.

Results. Our evaluation shows that the 319 samples exhibit the same file and network activities under VM Cloak, as when they are run in Oracles. This demonstrates that VM Cloak can be used to hide VMs efficiently. In our data set, 252 out of 319 (79%) samples show at least one anti-VM attack. The spectrum of the of anti-VM techniques are shown in Table X. For the semantic attack category, the most popular instruction is the `in` instruction. This instruction copies the value from the I/O port specified by the second operand (source operand) to the first operand (destination operand), and is usually used by VMs to set up the communication channel between the host and the guest system. Therefore, it behaves differently in a virtual and a physical machine and can be exploited by malware.



(a) Testing infrastructure (b) Execution Logic

Fig. 6. Testing Environment of VM Cloak

We observe that certain kernel registers are also popular in semantic attacks, such as local and global descriptor table registers, task register, and interrupt descriptor table register, which can be retrieved respectively using `sldt`, `sgdt`, `str`, and `sidt` instructions. This is because these registers are already set to fixed values by the host system, and VMs have to save these values and replace them with values needed by the VM. This behavior can be used by malware to detect VMs. The `cpuid` instruction returns a value that describes the processor features. After executing this instruction with `eax = 1`, the 31st bit of `ecx` on a physical machine will be equal to `0`, while this bit is `1` on a VM. The `smsw` instruction stores the machine status word (bits 0 through 15 of control register `cr0`) into the destination operand. Sometimes, the `pe` bit of `cr0` is not set in a VM. The `movdqa` instruction moves a double quad-word from the source operand to the destination operand. This instruction can operate on an XMM register and a 128-bit memory location, or between two XMM registers. The destination may be populated with a random value by a VM if the source operand is not available in certain cases, while the destination operand is untouched in a physical machine.

For string attacks, we find only one API (`RegEnumKeyExA()`) that is adopted by malware. This function enumerates the subkeys of the specified open registry key. In this case, malware attempts to find if "vmwa" exists, which is the prefix of a VM product – VMWare.

For timing attacks, we discover four APIs or instructions that malware uses to query date and time information. The prevalent API is `GetTickCount()`, which retrieves the number of milliseconds that have elapsed since the system was started, up to 49.7 days. Malware may call this API several times to check the elapsed time of executing certain code block. If the elapsed time exceeds a reasonable threshold, malware will detect a VM. This attack strategy can also be used to detect debuggers. Similarly, `QueryPerformanceCounter()` fetches the current value of the performance counter, which is a high resolution (1us) time stamp that can be used for time-interval measurements. The `rdtsc` instruction reads the timestamp counter register, and `GetLocalTime()` obtains the current local date and time. All these instructions can be exploited by malware.

Table X. Spectrum of Anti-VM Techniques

Category	Instruction	Samples	Instruction	Samples
Semantic	in	87/35%	smsw	16/6%
	sldt	65/26%	sgdt	10/4%
	str	49/19%	sidt	8/3%
	cpuid	35/14%	movdqa	6/2%
Category	**API/Instruction**		**Samples**	
String	RegEnumKeyExA(''vmwa'')		3/1%	
Timing	GetTickCount()		51/20%	
	QueryPerformanceCounter()		27/11%	
	rdtsc		8/3%	
	GetLocalTime()		2/1%	

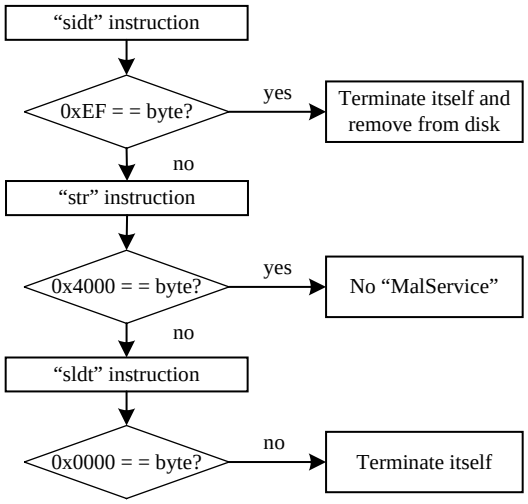


Fig. 7. Sample 1

6.4.4. VM Cloak Works: Testing with Known Malware. We now test if VM Cloak can detect anti-VM behaviors identified by other researchers. We collected three such samples, where ground truth is known for anti-VM techniques they use.

Sample 1 (md5: 6bdc203bdfbb3fd263dadf1653d52039). This sample is provided and analyzed by [30], and Figure 7 shows the anti-VM techniques that are employed. The sample employs three semantic attacks, including sidt, str, and sldt instructions. The sidt instruction stores the contents of idtr register into a memory location. The IDTR is 6 bytes, and the fifth byte offset contains the start of the base memory address. If this byte is equal to 0xef, the signature of "VMware" is detected. This sample will terminate itself and remove it from disk if this anti-VM check succeeds. To handle this attack, we overwrite the byte with the value of 0xff that is learned from an Oracle.

Similarly, str and sldt store the task register and the local description table register whose contents are different in a virtual machine. We update their values with those found in Oracles, so malware cannot detect the VMs.

Sample 2 (md5: 7a2e485d1bea00ee5907e4cc02cb2552). This sample has been analyzed by [30], and it uses one semantic attack and three string attacks, as shown in Figrue 8. The in instruction is a privileged instruction; it will raise an exception if administrator right is not granted. However, VMware uses virtual I/O ports for communication between the virtual machine and the host to support functionalities like copying and pasting between the two systems. The port can be queried and compared

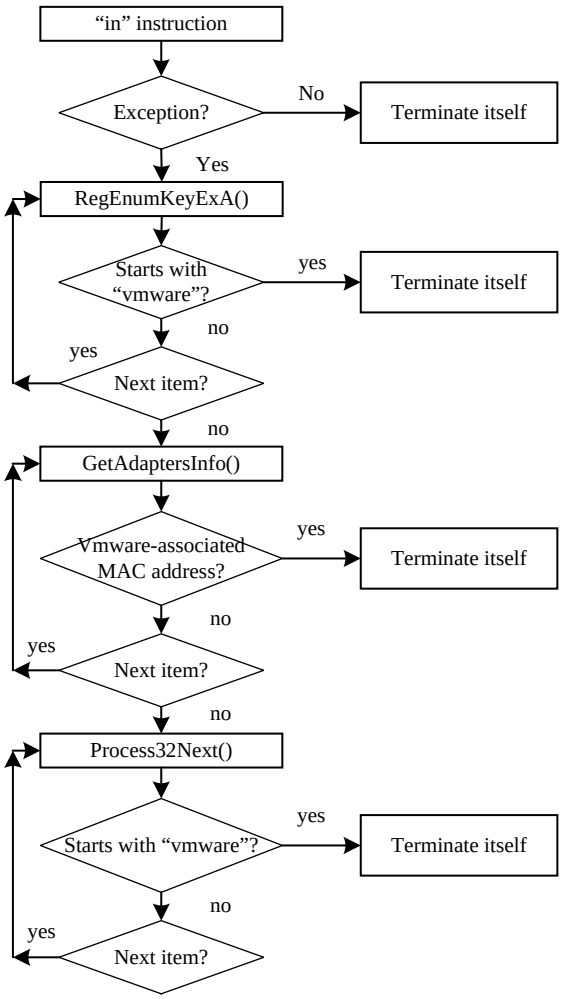


Fig. 8. Sample 2

with a magic number to identify the use of VMware by using `in` instruction. To handle this attack, we intentionally raise an exception after the execution of this instruction and overwrite the returned bytes with random ones.

Next, this malware uses three system APIs to query possible strings containing certain VM bands, such as “VMWare”. The `RegEnumKeyExA()` is used to enumerate all registry entries under “SYSTEM\\CurrentControlSet\\Control\\Device”. The sample compares the first six characters (after changing them to lowercase) of each subkey name to the string “vmware”. The `GetAdaptersInfo()` retrieves the MAC addresses of Ethernet or wireless interfaces. The addresses are then compared against known ones for VMware, such as “005056h” and “000C29h”. Finally, `ProcessNext32()` queries the names of all processes. For each process name, the malware hashes it into a number and then checks if this number is equal to the hash value of “vmware”. This, however, does not deter VM Cloak. We handle these string attacks, by monitoring the calls to the APIs and replacing the returned strings with random characters.

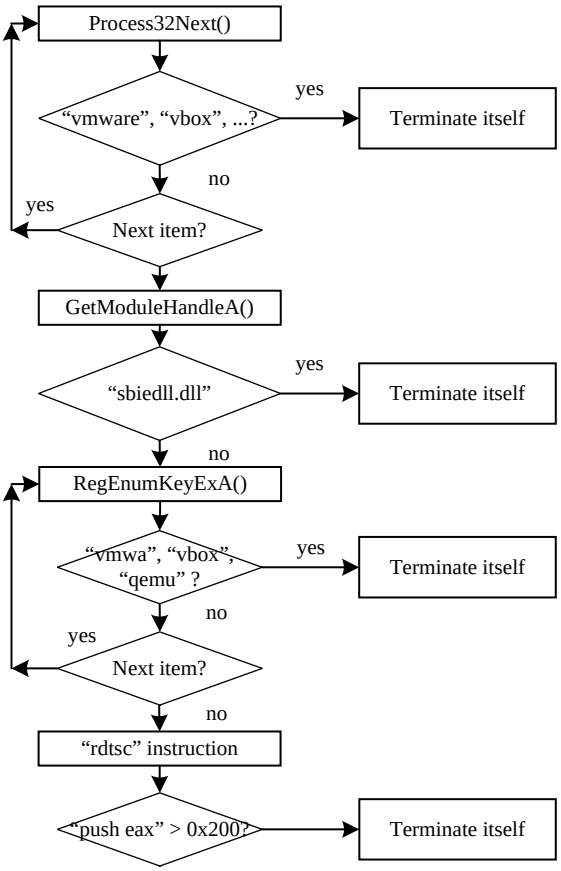


Fig. 9. Sample 3

Sample 3 (md5: `2c1a7509b389858310ffbc72ee64d501`). This sample is analyzed by [31], as shown in Figure 9. It performs both string and timing attacks. First, it checks the names of processes by comparing their CRC32-hashes to predefined ones of “vmwareuser.exe”, “vboxservice.exe”, “vboxtray.exe”, and many others. If this check passes, the sample will use `GetModuleHandleA()` to query the existence of “sbiedll.dll”, which is the artifact created by a VM – Sandboxie. The third string attack is reading the registry keys in “SYSTEM\\CurrentControlSet\\Services\\Disk\\Enum”. If any subkey string starts with “vmwa”, “vbox”, or “qemu”, the sample will terminate and exit. Finally, the sample uses two consecutive `rdtsc` instructions to measure the execution time for one `push eax` instruction. If the time exceeds `0x200`, the sample will exit immediately.

Results. We evaluated our VM Cloak by running these three selected samples under VMware and QEMU, with VM Cloak, and comparing the malware behavior in this environment with its signature behavior on an Oracle. We found that all three samples exhibited the same file and network activities when VMWare and QEMU were hidden by VM Cloak, as when the samples were run in Oracles. However, all three samples show early termination when executed in VMWare and QEMU without VM Cloak. We conclude that VM Cloak has successfully hidden VMWare and QEMU from malware.

Table XI. Performance Cost of VM Cloak

Testing Stage	Target	Category	Average Time Cost
Malware execution	Instruction	Disassemble	893 microseconds
		Match against attacks	11 milliseconds
		Modify program state if necessary	2 milliseconds
Disk analysis	Sample	Transfer image through network	21 minutes (8GB)
		Analyze image	26 minutes
Network analysis	Sample	Analyze network trace	3 seconds
System restore	Sample	Restore disk image	14 minutes

6.4.5. Performance Cost. Table XI shows the performance cost of VM Cloak in each testing stage (Figure 6b). During the analysis of each sample, we log the wall time at each stage and calculate duration of a stage as a difference between two adjacent timestamps. The values in the table are averaged over all the samples evaluated in the previous sections. During the malware execution stage, it takes on the average 893 microseconds to disassemble one instruction. This delay is caused by WinDbg itself, without our modifications. Next, we spend 11 milliseconds in matching the instruction against possible attacks, and 2 milliseconds in modifying program state, if necessary. Thus, VM Cloak is $13,000/893 = 14.6$ times slower than a vanilla WinDbg.

The performance cost is dominated by the disk image handling. For example, we need 21 minutes to transfer the image and require additional 26 minutes to extract file activities from the image. Furthermore, it takes 14 minutes to restore the disk image of MEE. Overall, the analysis delay adds up to 81 minutes (Table XI) for a single malware sample. In order to speed up the evaluation process, we launch five instances of infrastructure shown in Figure 6(a), to analyze malware samples in parallel. In this deployment, the amortized cost of analyzing one sample is approximately 16 minutes.

7. DISCUSSION

While our work addresses how to hide VMs from malware, there are several other approaches to malware analysis that do not use VMs. We now briefly discuss these other approaches. We note that there are two main challenges for malware analysis. First, it is expected that malware will contaminate the system and thus it is necessary to restore system states after each analysis cycle, such as disk content. Second, since one must instrument software or hardware to add analysis functionalities, it is critical to hide the introduced artifacts because anti-analysis malware can detect them and evade analysis.

7.1. OS Instrumentation

The most direct way is to design the analysis framework as an extension to the operating system that runs on bare-metals. However, it is difficult to handle both system restore and artifact hiding challenges in this type of instrumentation. For example, BareBox [**?**] proposes a malware analysis framework based on a fast and rebootless system restore technique. For memory isolation, they divide the physical memory into two partitions, one for the OS and the other for malware execution. For disk restore, the authors use two identical disks as main and mirror configuration. When saving a snapshot, they redirect all write operations to the mirror disk, so the contents of the main disk will be effectively frozen. While these techniques help restore the system within a few seconds, BareBox has very limited approaches to artifact hiding. For example, malware can perform string attack, e.g., enumerating process names to match against "BareBox". Since BareBox runs at the same privilege level as the OS, malware running at ring 0 can always detect BareBox.

Popular malware-oriented debuggers [**?**; **?**; **?**] can also be classified into this category. These frameworks propose a debugging strategy that aims to analyze malware in a

fine-grained, transparent, and faithful way. However, it is not an easy task to hide debuggers from malware, and malware authors have devised a variety of anti-debugging techniques. Generally, malware can attack the debugging principles or the artifacts introduced by debuggers. For example, in order to set up a software breakpoint at an instruction, the debuggers need to replace the beginning opcode of the instruction with a `0xcc` byte. This will raise a breakpoint exception upon execution, which will be captured by the debuggers. To perform a breakpoint attack, malware may scan its opcode for the special byte, or calculate the hash value of its opcode and compare it to a predefined value. In addition, malware can also attack debuggers by detecting their exception handling, flow control, disassembling, and many other principles used for analysis. The focus of VM Cloak is to address the virtualization transparency problem. Our other related work – Apate [**?**] – addresses anti-debugging and can be easily integrated with VM Cloak.

7.2. Bare-metal Instrumentation

This category of instrumentation introduces the fewest artifacts among the malware analysis methods, by instrumenting on-board hardware with analysis functionalities. Spensky et al. [**?**] (LO-PHI) modify a Xilinx ML507 development board, which provides the ability to passively monitor memory and disk activities through its physical interfaces. Since they do not use any VMs or debuggers, there are no artifacts at that level that a malware may detect. However, malware could attempt to detect presence of this particular development board and avoid it, assuming that it is used for analysis.

BareCloud [13] replaces analysis of local disk by using the iSCSI protocol to attach remote disks to local system. After each run of malware, the authors extract file activities from the remote disk and restore it through copy-on-write technique. While this disk restore method improves system recovery efficiency, the actual evaluation overhead is not mentioned in the paper. The system management mode (SMM) is used by Zhang et al. [28] to implement debugging functionalities. SMM is a special-purpose CPU mode in all x86 processors. The authors run malware on one physical target machine and employ SMM to communicate with the debugging client on another physical machine. While SMM executes, Protected Mode is essentially paused. The OS and hypervisor, therefore, are unaware of code execution in SMM. However, MalT is designed to run on a single-core system, which will be problematic in a multi-core environment. The authors argue that MalT can debug a process by pinning it to a specific core, while allowing the other cores to execute the rest of the system normally. This will change thread scheduling for the debugged process by effectively serializing its threads, and can be used by malware for detection.

Even without any instrumentation, Miramirkhani et al. [**?**] demonstrate that malware can still detect analysis systems using wear-and-tear artifacts. In this work, the authors extract the system artifacts that occur in daily use but not in a sandbox. For example, browser will exhibit certain diversity in the URLs visited, such as games, e-shopping, and e-banking, which current analysis systems do not mimic. While these artifacts can be used to detect analysis systems with high accuracy, our work focuses on an orthogonal problem of hiding VMs from malware.

Ninja [**?**] is a transparent malware analysis framework on ARM platform. Ning and Zhang utilize the hardware-assisted isolation execution environment, TrustZone, to achieve a high level of transparency. Our work is complementary to theirs. Ninja requires special hardware (with TrustZone), while our approach works on any hardware. On the other hand, our Cardinal Pill Testing may conceivably miss some traces left by VMs that malware could find. Ninja bypasses this problem by not using a VM for malware analysis.

8. CONCLUSION

Virtualization is crucial for malware analysis, both for functionality and for safety. Contemporary malware aggressively checks if it is being run in VMs and applies evasive behaviors that hinder its analysis. Existing works on detection and hiding of differences between virtual and physical machines apply ad-hoc or semi-manual testing to identify these differences and hide them from malware. Such approaches cannot be widely deployed and do not guarantee completeness.

In this paper, we first propose cardinal pill testing that requires moderate manual action per CPU architecture, to identify ranges for input parameters for each instruction. It then automatically devises tests to enumerate the differences between any pair of physical and virtual machines. This testing is much more efficient and comprehensive than state-of-the-art red pill testing. It finds five times more pills running fifteen times fewer tests. We further claim that for instructions that affect defined resources, cardinal pill testing identifies all possible test pills, i.e., it is complete. Other categories contain instructions whose behavior is not fully specified by the Intel manual, which has led to different implementations of these instructions in physical and virtual machines. Such instructions need understanding of the implementation semantics to enumerate all the pills and devise the hiding rules. However, these pills cannot be exploited by attackers because they are not persistent.

Next, we propose a pill hiding framework called VM Cloak – a debugger plug-in that examines each instruction for possible anti-VM checks, and overwrites register and memory states with expected values to hide VM presence. We implement VM Cloak as a WinDbg plug-in and show through small-scale evaluation that it successfully hides VMware and QEMU from malware. We believe that cardinal pill testing and VM Cloak have many advantages – they are agnostic to VM and physical machine choices, enable comprehensive detection and hiding of VM checks, and are easily adoptable.

REFERENCES

L. Martignoni, R. Paleari, G. F. Roglia *et al.*, "Testing CPU Emulators," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

P. Barford and M. Blodgett, "Toward Botnet Mesocosms," in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

J. P. John, A. Moshchuk, S. D. Gribble *et al.*, "Studying Spamming Botnets Using Botlab," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

C. Kreibich, N. Weaver *et al.*, "GQ: Practical Containment for Measuring Modern Malware Systems," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, 2011.

L.-K. Yan, M. Jayachandra, M. Zhang *et al.*, "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)*, 2012.

C. Song, P. Royal, and W. Lee, "Impeding Automated Malware Analysis with Environment-Sensitive Malware," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec)*, 2012.

U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

"BitBlaze: Binary Analysis for Computer Security," http://bitblaze.cs.berkeley.edu/.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

K. P. Lawton, "Bochs: A Portable PC Emulator for Unix/X," *Linux Journal*, no. 29es, 1996.

P. Ferrie, "Anti-Unpacker Tricks," http://vpn23.homelinux.org/Anti-Unpackers.pdf.

——, "Attacks on Virtual Machine Emulators," *Symantec Security Response*, 2006.

D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in *23rd USENIX Security Symposium*, 2014.

R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies," in *Black Hat*, 2012.

M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting Environment-Sensitive Malware," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

X. Chen, J. Andersen, Z. Mao *et al.*, "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware," in *IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, 2008.

L. Martignoni, R. Paleari, G. Fresi Roglia *et al.*, "Testing System Virtual Machines," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

M. G. Kang, H. Yin, S. Hanna *et al.*, "Emulating Emulation-resistant Malware," in *Proceedings of the First ACM Workshop on Virtual Machine Security (VMSec)*, 2009.

H. Shi, A. Alwabel, and J. Mirkovic, "Cardinal Pill Testing of System Virtual Machines," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 337–348.

A. Dinaburg, P. Royal *et al.*, "Ether: Malware Analysis via Hardware virtualization Extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

G. Pék, B. Bencsáth, and L. Buttyán, "nEther: In-guest Detection of Out-of-the-guest Malware Analyzers," in *Proceedings of the Fourth European Workshop on System Security (EuroSec)*, 2011.

F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX ATC*, 2005.

M.-K. Sun, M.-J. Lin, M. Chang *et al.*, "Malware Virtualization-Resistant Behavior Detection," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

D. Balzarotti, M. Cova, C. Karlberger *et al.*, "Efficient Detection of Split Personalities in Malware," in *Network and Distributed System Security (NDSS)*, 2010.

Intel, "Intel 64 and IA-32 Architectures Software Developers Manuals," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

R. Bajcsy, T. Benzel, Bishop *et al.*, "Cyber Defense Technology Networking and Evaluation," *Commun. ACM*, vol. 47, no. 3, 2004.

F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using Hardware Features for Increased Debugging Transparency," in *Proceedings of The 36th IEEE Symposium on Security and Privacy*, May 2015.

B. Technology, "The Sleuth Kit," http://www.sleuthkit.org/.

M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

0xEBFE, "Fooled by Andromeda," http://0xebfe.net/blog/2013/03/30/fooled-by-andromeda/.