Cardinal Pill Testing of System Virtual Machines

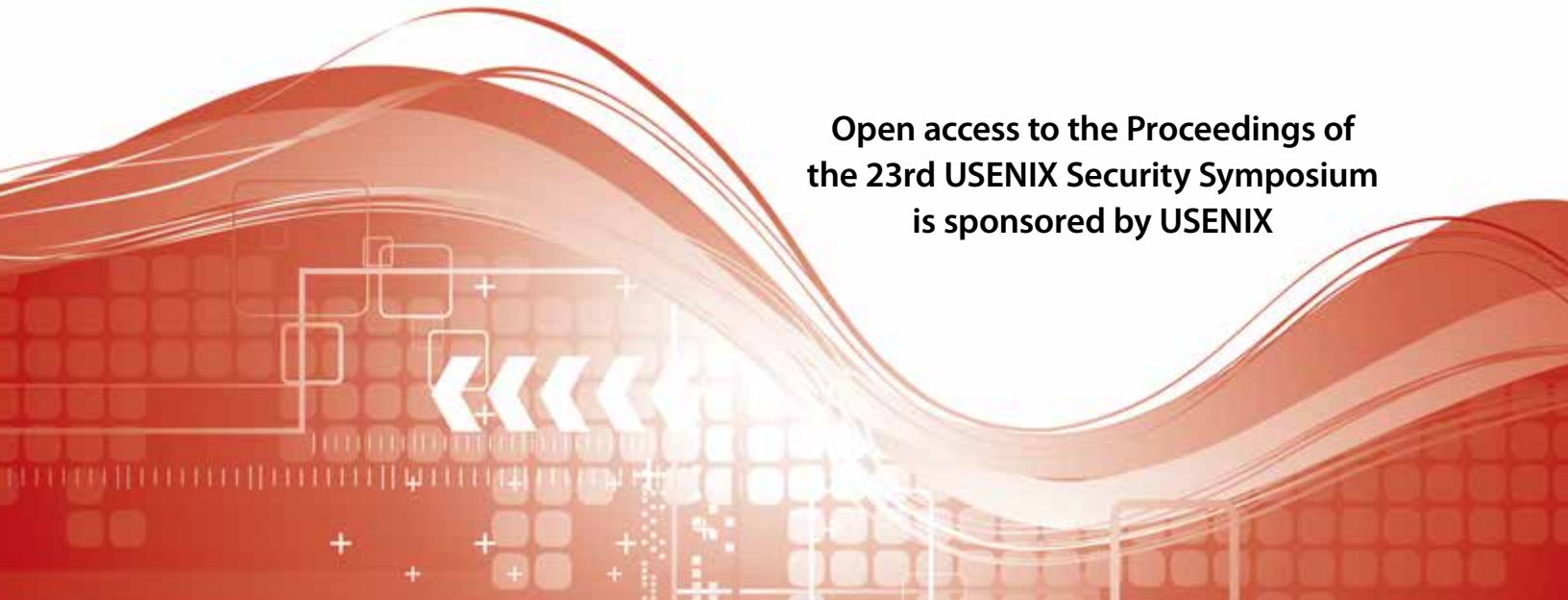
Hao Shi, Abdulla Alwabel, and Jelena Mirkovic, ***USC Information Sciences Institute (ISI)***

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/shi

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7



Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

Cardinal Pill Testing of System Virtual Machines

Hao Shi
USC/Information Sciences Institute
haoshi@usc.edu

Abdulla Alwabel
USC/Information Sciences Institute
alwabel@usc.edu

Jelena Mirkovic
USC/Information Sciences Institute
mirkovic@isi.edu

Abstract

Malware analysis relies heavily on the use of virtual machines for functionality and safety. There are subtle differences in operation between virtual machines and physical machines. Contemporary malware checks for these differences to detect that it is being run in a virtual machine, and modifies its behavior to thwart being analyzed by the defenders. Existing approaches to uncover these differences use randomized testing, or malware analysis, and cannot guarantee completeness.

In this paper we propose Cardinal Pill Testing—a modification of Red Pill Testing [21] 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 further examine the causes of pills and find that, while the majority of them stem from the failure of virtual machines to follow CPU design specifications, a significant number stem from under-specification of the effects of certain instructions by the Intel manual. This leads to divergent implementations in different CPU and virtual machine architectures. Cardinal Pill Testing successfully enumerates differences that stem from the first cause, but only exhaustive testing or an understanding of implementation semantics can enumerate those that stem from the second cause. Finally, we sketch a method to hide pills from malware by systematically correcting their outputs in the virtual machine.

1 Introduction

In today's practice of analyzing malware [3, 14, 16, 26, 23], system virtual machines are widely used to facilitate fine-grained dissection of malware functionalities (e.g., Anubis [4], TEMU [6, 24], and Bochs [17]). For example, virtual machines can be used for dynamic taint analysis, OS-level information retrieval, and in-depth behavioral analysis. Use of virtual machines also protects the host, by isolating it from potentially malicious actions.

Malware authors have devised a variety of methods to hinder automated and manual analysis of their code, such as anti-dumping, anti-debugging, anti-virtualization, and anti-intercepting [10, 11]. Recent studies [7, 18] show that anti-virtualization and anti-debugging techniques have become the most popular methods of evading malware analysis. Chen et al. [8], 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. [18] detect evasion behavior in 25.6% of 1,686 malicious binaries. In 2012, Branco et al. [7] analyze 4 million samples and observe that 81.4% of them exhibit anti-virtualization behavior and 43.21% exhibit anti-debugging behavior.

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. 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, and specifically on CPU semantic attacks.

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 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.

These attacks are successful because there are many differences between VMs and physical machines, and existing research in VM detection [21, 20, 15] uses ad-hoc tests that cannot fully enumerate these differences. Since 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 could use this database to provide expected replies to malware queries, thus hiding its presence. This is akin to kernel root kit functionality, where the root kit 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 attempt 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 are:

1. We improve on the previously proposed Red Pill Testing [21, 20] 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.

3. We propose how to modify virtual machines to automatically hide presence of detected pills from malware, through introduction of additional interrupt vectors and by utilizing QEMU's interrupt handling mechanism for guest systems (Tiny Code Generation mode).

We emphasize that our testing methodology produces test cases selected at random from chosen input parameter ranges for each instruction – these ranges are chosen to exercise all execution paths in the given instruction's handling. If a test case's execution produces different outputs in a physical versus a virtual machine we say that this test case is a *pill*. While we only test one value from each parameter's range, if this test case is a pill, all values from the same parameter ranges would also lead to pills because they are all handled by the same path in that instruction's execution. Let us call a pill resulting from a test case a *test pill* and all related test cases that draw parameter values from the same input ranges as the test pill the *individual pills*. In this paper, all counts of pills we report are for test pills. Similar practice is adopted by related work [21, 20, 19]. The counts of individual pills are many times higher.

In Section 2 we give an overview of various anti-virtualization techniques. We survey related work in Section 3 and propose Cardinal Pill Testing in Section 4. We provide the details for the pills we find in Section 5 and analyze their root causes and completeness. In Section 6 we propose how to hide most of these pills from malware and we conclude in Section 7. All the scripts and test cases used in our study are publicly released at `http://steel.isi.edu/Projects/cardinal/`.

2 Anti-Virtualization Techniques

Anti-virtualization techniques can be classified into the following broad categories [8, 15]:

CPU 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 [9] 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 presence of a VM.

Timing Attacks. Malware measures the time needed to run an instruction sequence, assuming that an operation takes a different amount of time in a virtual machine compared to a physical machine [11]. Contemporary virtualization technologies (dynamic translation [5], bytecode interpretation [17], and hardware assistance [9]) all add significant delays to instruction execution that are measurable by malware [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 would reveal the VM's presence immediately [8].

In this work we focus on handling the CPU semantic attacks as they are the most complex category to explore and enumerate. We note that string attacks can easily be handled through enumeration and hiding of VM traces, which can be done by comprehensive listing and comparison of files, processes and Windows registry with and without virtualization. Also, timing attacks can be handled through systematic lying about the VM clock, as proposed in [15]. While neither of these approaches

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

is implemented today, both could be implemented as extensions of our work on lying to applications about CPU semantics (Section 6).

3 Related Work

Martignoni et al. present the initial Red Pill work in EmuFuzzer [21]. They propose Red Pill Testing—a method that performs a random exploration of a CPU instruction set and parameter spaces looking 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 KEmuFuzzer [20], Martignoni et al. extend the state definition to include the kernel space memory, and test cases are embedded in the kernel to facilitate testing of privileged instructions. In their recent work [19], 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.

While these works are seminal in pill detection they have several deficiencies that we seek to handle in this paper: (1) EmuFuzzer [21] tests boundary and random values for explicit input parameters, but does not cover implicit parameters. Their approach cannot guarantee that all types of pills will be detected. The symbolic execution approach [19] will discover differences between low-fidelity and high-fidelity emulators but not between an emulator and a physical machine. In addition, use of symbolic execution precludes test generation for floating-point instructions. We improve on these works by using instruction semantics to carefully craft test cases that explore all code paths. (2) Martignoni et al. use QEMU with Intel VT-x (in [21]) or Bochs emulator (in [19]) as an Oracle, while we use physical machines with no virtualization. This improves fidelity of testing and ensures detection of more pills.

Dinaburg et al. [9] 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. nEther [22] work finds that Ether still has significant differences in instruction handling when compared to physical machines, and thus anti-virtualization attacks are still possible, i.e., Ether does not achieve complete transparency.

Kang et al. [15] propose a method to identify anti-emulation checks and modify virtual system states to "lie" to the malware, using semi-manual execution trace analysis. They record the malware trace in Ether, using it as an Oracle, and utilizing its debugging functions. They then automatically taint the variables in this trace, and manually identify those variables whose values are used in an anti-emulation check under QEMU. Their method requires manual intervention while we seek to overcome differences in execution environments automatically. Furthermore, since Ether is not identical to a physical machine, this approach will fail to detect some differences between a VM and a physical machine that we do detect.

Other works [25, 18, 2] focus on detecting anti-virtualization functions of malicious binaries based on profiling and comparing their behavior in virtual and physical machines. These works 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 used in future anti-virtualization checks.

4 Cardinal Pill Testing

We now describe the architecture, test case generation and testing methodology for our Cardinal Pill Testing.

4.1 Architecture Overview

Our testing architecture is shown in Figure 1. It consists of three physical workstations: a master, a slave hosting a virtual machine (VM), and a slave running Windows 7 Pro x86 on a bare-metal as reference (Oracle). The slaves are connected to the master through two separate serial wires. The master is responsible for generating test cases (Section 4.3) and scheduling their execution in slaves. In both slaves, we configure an additional daemon in the testing system that helps the master set up a specific test case in each testing round.

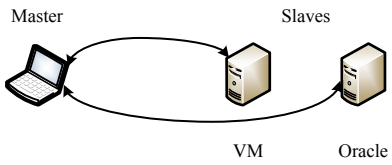


Figure 1: Architecture Overview

4.2 Logic Execution

The execution logic of our Cardinal Pill Testing is illustrated in Figure 2. 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 daemon; we only show one pair of test case

and daemon in Figure 2 for clarity. We set the slaves in kernel debugging mode so that they can be completely frozen when necessary. 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 by one in terms of rounds.

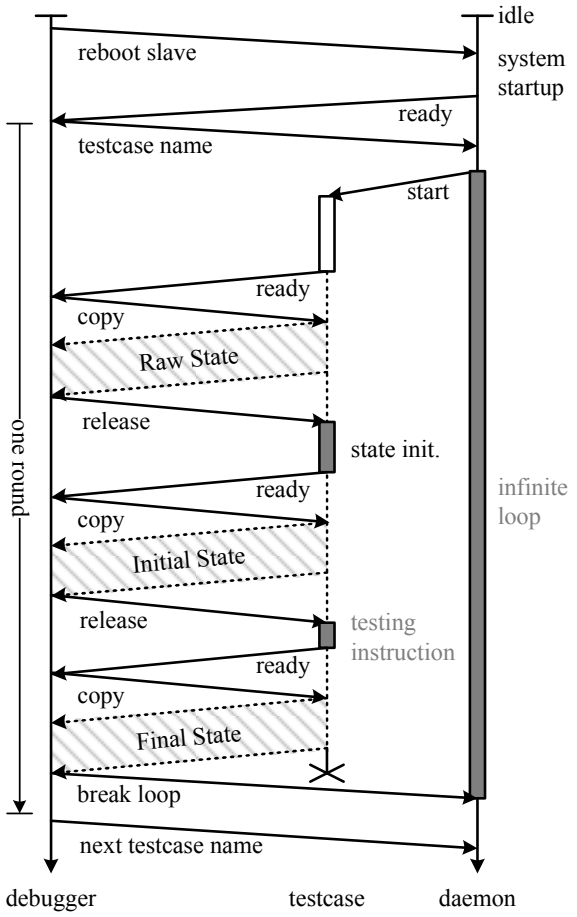


Figure 2: Logic Execution

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 that our test case accesses for reading or writing.

In each round, the master interacts with the slave in three main phases. In the first phase, it issues a test case name to the daemon, which resides in a slave, and the daemon will ask the slave system to load this test case stored in its local disk. Then the system starts allocating memory, handles, and other resources needed by the test case program. After this *system loading* completes, the test case executes an interrupt instruction (`int 3`), which notifies the master and halts the slave. At this moment, the master saves the *raw state* of the slave locally. We use this raw state to identify *axiom pills* (see Section 4.3), instead of discarding it, as is done by EmuFuzzer [21] and KEmuFuzzer [20].

In the second phase, the master releases the slave which then executes the test case's initialization code and raises the second interrupt. Instead of using the same initial system state for all test cases, we carefully tailor register and memory bits for each test case, such that all possible exceptions and semantic branches can be evaluated (see Section 4.3). The master copies back 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). It may happen that a test case drives the slave into an infinite loop or crashes itself or its OS. To detect this, we set up an execution time limit for each test case, so that the master can detect incapacitated slaves and restore them.

4.3 Test Case Generation

The quality of test cases is the key component of efficient pill discovery. The Red Pill work [21] generates test cases via two approaches: random generation and CPU-assisted generation. The former method randomizes data and code without conforming to any semantic rules, which may encode invalid instruction sequences. The latter combines each known opcode with some predefined operand values. Both approaches have the following deficiencies: (1) They only consider operands encoded in the instruction and fail to consider implicit arguments whose value may lead instruction execution to a different path in the code. For example, `rep stosb` takes no arguments but it depends on multiple register values. It stores contents of `al` register at the address specified by `es:(e)di`, and does this `ecx` times. Different values placed into those registers will result in different scenarios for `rep stosb` command use, such as writing into a valid versus invalid memory location, overwriting the instruction itself, using a zero, negative or very large positive value for the number of repetitions, etc. (2) They generate operands for instructions at random, which also does not explore all possible code paths. Our test case generation algorithm addresses both of these challenges.

4.3.1 Testing Goals

We aim to generate a minimal set of test cases for each instruction that explore all possible code paths in this instruction's handling. We start from the definitions of instruction handling recorded in a CPU manual. In this

work we focus on Intel's x86 CPU processor [13]. The manual details inputs and outcomes for each instruction, for normal execution and for exception handling. We call register modifications and exceptions whose semantics are fully defined in the manual *defined behaviors*. An instruction may also affect registers and raise exceptions that are specified in the manual as affected but the manner of their modification by the instruction is not specified. We call these modifications *undefined behaviors*.

For example, the `aaa` instruction adjusts the sum of two unpacked binary coded decimal (BCD) values to create an unpacked BCD result. The `al` register is the implied source and destination operand for this instruction. It also reads the `AF` flag in the `EFLAGS` register and writes to the `ah` register. Its normal execution will set `AF` and `CF` flags to `1` if the adjustment results in a decimal carry; otherwise it will set them to `0`. This is the defined behavior for the `aaa` instruction. In our testing we find that physical machines also set or reset the `SF`, `ZF`, and `PF` flags. While these flags are listed as affected by the instruction in the manual, there are no details of how they are calculated or what semantics they carry for the `aaa` instruction. This is the undefined behavior for the `aaa` instruction. In our work we explore both defined and undefined behaviors for each instruction, because both of these can be the source of pills.

Based on these observations, we set up the following goals of our test case generation algorithm:

- For defined behaviors for a given instruction, all branches should be evaluated. All flag bit states that are read implicitly or updated using results must be considered.
- All potential exceptions must be raised, such as memory access and invalid input arguments.
- Undefined behaviors should be investigated to reveal undocumented implementation specifics.

In the following sections, we first illustrate our test case template and then discuss how we group instructions and populate the template.

4.3.2 Test Case Template

We program a template to automatically generate test cases for most instructions, as shown in Figure 3. This program notifies the master 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 ignore Windows' built-in exception handling routine and jump to line 31 directly, so we can save the system state before exception handling.

```
 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
```

Figure 3: Test Case Template (in MASM assembly)

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. After this, 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 29, where all defined and undefined behaviors use will be evaluated in various test cases. When compiling test cases, we disable

linker optimization and use a fixed base address, which does not affect testing but eases the interaction between the master and slaves.

4.3.3 Instruction Grouping

To test each instruction's behaviors in different execution stages, we need to vary the content in all registers and memory that the instruction reads. As discussed earlier and demonstrated in our evaluation in Section 5, random test generation cannot guarantee coverage of all code paths and execution branches. In our method, we manually analyze instruction execution flows defined in Intel manuals [13] and classify all possible input parameter values into ranges that lead to distinct execution flows.

IA-32 CPU architecture contains 906 instruction codes, and a human must reason about each to identify its inputs and outputs and how to populate them to test all execution behaviors. To reduce the scale of this human-centric operation, we first group the instructions into five categories: arithmetic, data movement, logic, flow control and miscellaneous. Arithmetic and logic category are further 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 chance of human errors. It took one person on our team a month and a half to devise all test cases. Table 1 shows the number of different mnemonics, examples, and parameter ranges we evaluate for each category.

Arithmetic Group. Instructions in this group first fetch arguments and then perform arithmetic operations. The arguments include actual data bits they operate on and certain flag bits that decide execution branches. We classify instructions in this group into two subgroups, depending on whether they work only on integer registers (general-purpose group), or also on floating point registers (FPU group). The instructions in the FPU group include instructions with x87 FPU, MMX, SSE, and other extensions.

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 flip `AF` between clear and set 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 without required privileges, and into a segment that is paged in and 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 copy data between registers, main memory, and peripheral devices and usually do not modify flag bits. There are several execution branches that we explore in tests. The source and the destination operands may be located outside segment limits. If the effective address is valid but paged out, a page-fault exception will be thrown. If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3, the system will raise an alignment exception. Some instructions also check direction and conditional flags, and a few others validate the format of floating point values. All these input parameters and the states that influence an instruction's execution outcome must be tested.

For example, we group 30 conditional movement instructions `cmov`*`cc`* `r32, r/m32` of distinct *`cc`* together because they move 32 bit signed or unsigned integers from the second operand (32 bit register or memory) to the first operand (32 bit register). The *`cc`* conditions are determined by the `CF`, `ZF`, `SF`, `OF` and `PF` flags. To access arguments outside the segment limit, we compile our test cases with the fixed base (Section 4.3.2). The starting addresses for code, data, and stack segment are `401000h`, `403000h`, and `12e000h` respectively, and each has a size of 4KB. It is difficult to test page faults

Category	Instruction 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

Table 1: Instruction Grouping

directly because the Windows system does not provide APIs for page swapout. To work around this, we run other memory-consuming programs between test cases that use memory operands to force the values to be paged out of memory. In our evaluation, we find that this strategy works well and we successfully raise page faults when we need to test them. To raise the alignment checking exception, we store instruction operands at unaligned memory addresses. We permute the condition bits in the same way as we do for testing of arithmetic instructions.

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 further partition logic instructions based on the flag bits they read and argument types and sizes. When designing test cases, in addition to testing minimal, maximal, and boundary values for each parameter, for instructions that compare two parameters we also generate test cases where these parameters satisfy larger-than, equal, 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 and greater-than relationships. While the operand relationship does not influence instruction execution in this case, it does for other subgroups, e.g. the one containing the cmp instruction.

In the FPU subgroup, we apply similar rules to generate floating point operands. We further 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 different values into 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 non-conforming 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.

5 Detected Pills

We detect pills using our implementation of the architecture shown in Figure 1. 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), QEMU (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 Core i7 CPU and installs WinDbg 6.12 to interact with the slaves. For test case compilation, we use Microsoft Assembler 10 and turn off all optimizations. Our test cases take around 10 seconds to run on a physical machine and 15–30 seconds to run on a virtual machine.

Counting the different addressing modes, there are 1,653 instructions defined in the IA-32 Intel manual [13]. Out of these, there are 906 unique mnemonics. We generate a total of 19,412 test cases for these instructions.

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, we do not populate kernel registers in the initialization step because this may crash the system or lead it to an unstable status. Further, initialization of kernel registers would require a system reboot and would make testing prohibitively expensive in a virtual machine. But, kernel register contents 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 user memory are the same in the Oracle and the virtual machine in the initial state. If they are different, it means that the VM fails to virtualize the initialization instructions (line 26 in Figure 3) to match their implementation in the Oracle. We mark this test case as "fatal" and discard it. 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 in two scenarios: (2) when the user registers, exception registers, and memory in the final states are different and (2) when the values in a certain kernel register are the same in the initial states but different in the final states.

5.2 Results

Table 2 shows the results of comparing various virtual machines to Oracle1 (O1).

The second column shows the number of pills for different virtual machines. 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 our 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 assistance. 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 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 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 we find no fatal test cases for VT-x technology.

Table 3 shows the breakdown of pills per instruction

VMs	pills	crash	fatal
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%

Table 2: Results Overview

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

Table 3: Pills per Instruction Category

category from Figure 1. The FPU arithmetic, FPU logic and data movement categories contain the most pills—around 83%. Table 4 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.

5.2.1 Comparison with EmuFuzzer Pills

EmuFuzzer [21] generates 3 million test cases and the authors select 10% randomly to test in different virtual machines. The authors publish 20,113 red pills for QEMU 0.14.0-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, but instead we only compare the pills found by them and by us.

A *unique pill* is a pill whose mnemonic and parameter values do 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. After removing duplicate pills, there are 1,850 unique red pills (9%) and 136 different instruction mnemonics found by EmuFuzzer. Our 9,255 pills for Q1 (TCG) are all unique and there are 630 different instruction mnemonics. Furthermore, 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%. While direct comparison between our pills and EmuFuzzer's is difficult because both approaches select values of operands to test at random from specific ranges, we compare the ranges of the pills. This comparison shows that we detect all types of pills found by EmuFuzzer.

We conclude that our approach is more comprehensive than EmuFuzzer's and far more efficient. We cover all instruction mnemonics in our tests and find pills for 494 more instructions than EmuFuzzer. Overall we find five times more pills running 300,000/19,412 = 15 times fewer tests than EmuFuzzer. This illustrates the significant advantage of careful generation of operand values in tests over random fuzzing.

We further wanted to compare our pills with pills found by [19]. The Hi-Fi tests for Lo-Fi emulators [19] generate 610,516 test cases, out of which 60,770 (9.95%) show different behaviors in QEMU, and 15,219 (2.49%) show different behaviors in Bochs. Since the tests used for [19] are not publicly released we could not compare against them.

5.2.2 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 [13]. 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` and `OF`, `SF`, `ZF`, and `PF` flags are undefined.

Table 5 shows the number of pills that result from dif-

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

Table 4: 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

ferences 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 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, we analyze their root causes and compare them against those found by the symbolic execution method [19]. We find all root causes listed in [19] that are related to general-purpose instructions and QEMU's memory management unit.

In this work we do not extensively analyze pills that originate from differences in kernel-space handling of instructions, and thus cannot compare their root causes with those specified in [19]. Due to the extensive time required for testing (reboot is required after each test case) we leave this for future work.

Because the symbolic execution engine in [19] does not support FPU instructions, we discover additional root causes that are not captured by their 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. The details of all these root causes are given on our Web page.

5.2.3 Identifying Persistent Pills

Differences found in our tests between an Oracle and a virtual machine may not be present if we use 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. 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 differentiate between any physical and any virtual machine 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 [1]. Out of 195+23 = 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

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

Table 5: Pills using Undefined/Defined Resources

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		

Table 6: Undefined EFLAGS Behaviors

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.

5.2.4 Completeness of Pills

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 user-space instructions and part for kernel instructions defined in Intel manuals. Our test pills should thus include all possible individual pills that can be detected for defined resources in user space. We cannot claim the same completeness for test pills that relate to defined or undefined resources in kernel space since we do not extensively test instructions that manipulate these resources, due to the reboot requirement.

We now further 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. The only undefined resources from the Intel manual are flags in the EFLAGS register.

We analyze the user-space 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 6 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 6, 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 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, user-space resources. We can claim that because we fully understand semantics of these modifications, and all physical machines we tested strictly adhere to this 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. We further cannot claim completeness for pills that relate to modifications of kernel-space resources because we do not properly test initialization of these resources – such testing would require frequent reboots and would significantly prolong testing time.

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 7. For example, the value of `0ffffffffh` in the `edx` register can be used to diagnose the presence of Q2 (VT-x).

6 Improving Virtualization Transparency

EmuFuzzer [21] defines the virtualization transparency as how closely a virtual machine resembles the physical one. A perfect transparency means that programs in guests must not be able to tell if they are being executed in a virtual machine or not. The pills we find reflect the flaws of current virtual machine implementations, and specifically persistent pills reflect persistent flaws that can be used effectively by malware to detect virtualization. It would thus be desirable to develop techniques that hide the presence of reliable pills from malware. This could be achieved via multiple ways: (1) through patching of the current virtual machine implementations, (2) through overwriting of values in registers and memory with values consistent with physical machine deployment using kernel debuggers, (3) through modification of the guest OS so that malware reads of registers and memory after execution of pill instructions are intercepted and values consistent with physical machine deployment are returned (similar to kernel rootkit functionality), (4) through modifications of the host OS.

Out of all these approaches, patching VMs or guest OS are both time-consuming, may introduce other pills or bugs and do not apply to closed source implementations. Modifications to host OSes cannot hide all pills; for example in the TCG mode of QEMU, guest code translation happens in QEMU's user space, and the host cannot directly inspect guest instructions to detect pill execution. We thus choose to overwrite registers and memory

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

Table 7: Axiom Pills

after pill instructions.

This overwriting can either happen in the virtual machine, through modification of VM code, or it could be performed by the same environment that is used for malware analysis, e.g. Anubis or Ether. We explore the first strategy here. We select QEMU TCG mode as our experiment platform since it has gained great popularity [6, 24, 4, 12]. We first explain how QEMU handles guest code translation and then describe how we integrate our pill hiding strategy into its translation code.

6.1 The Underhood of QEMU with TCG

Figure 4 describes two pivotal functionalities of QEMU: how TCG uses translation blocks to organize translated host code (x86 guest to x86_64 host in the example) and how QEMU executes translation blocks. A translation block is a consecutive memory of a few kilobytes located in a data segment, which consists of translated host code, prologue, and epilogue. It provides a full function layout as if generated from a compiler. As the name implies, the translated host code section stores host opcode generated by TCG, which acts as a function body. The prologue prepares the stack and registers for use within the function, while the epilogue restores the stack and registers to the state they were in before the function was called.

TCG translates guest instructions in two different ways. Simple guest instructions are mapped to host opcode directly; for example in Figure 4, the guest instruction `mov al, 8` is transformed to three host instructions. The actual translation operates at the opcode level without disassembling and compilation. For complex guest instructions, TCG uses helper functions to implement their semantics. For example, the guest `int` instruction will be replaced by a call to `helper_raise_int()`. Inside this function, QEMU checks the current CPU mode and then dispatches the interrupt. In dispatching, QEMU calculates the destination vector in the interrupt description table that should be selected. After the desired interrupt service routine is found, QEMU sets the guest code segment selector, offset, and instruction pointer, such that the guest will enter interrupt handling immediately after QEMU

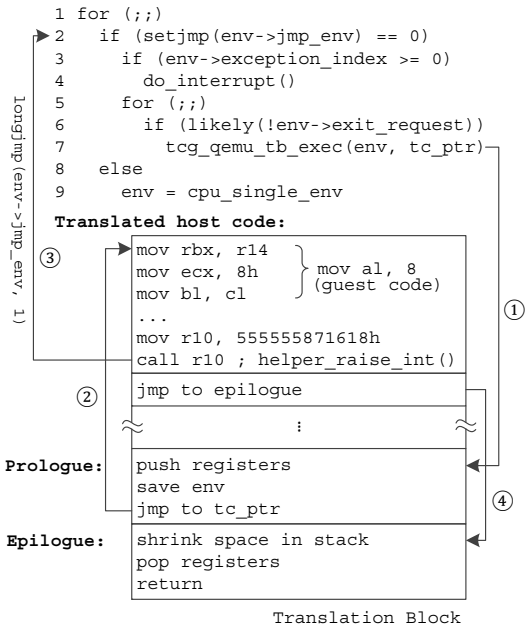


Figure 4: QEMU with TCG Translation and Execution.

yields control to the guest. Typically, QEMU stops translation if it encounters an `int` in the guest code, and `helper_raise_int()` will be the last instruction in a translation block in this case.

We summarize QEMU's main execution loop in lines 1–9 in Figure 4. It attempts to deliver all pending interrupts and exceptions and then finds the next translation block to execute. It takes advantage of the `setjmp()` and `longjmp()` facility provided by the C standard library to implement non-local jumps. At line 2, the QEMU context is saved to `jmp_env` by `setjmp()`. If this statement is actively called in place, line 3 will be examined and any pending interrupts and exceptions will be handled here. Otherwise, if the program flow returns here from a `longjmp()`, line 9 will be executed to reload CPU environment; then line 1 starts the next iteration. Lines 5–7 denote an infinite loop inside which QEMU repeatedly finds and executes translation blocks if no exception occurs. The function at line 7 is defined as a function pointer that is assigned to the

memory address of the prologue. At run-time, the prologue is cast as a function and executed (①) with parameters `env` and `tc_ptr`. The code bytes in the prologue save the current context and arguments to the stack. Then the program control will be transferred to the generated host code pointed to by `tc_ptr` (②). If the guest code contains an interrupt, the execution flow will follow the `helper_raise_int()` function generated by TCG (③); otherwise, this translation block will finish execution and step ④ is selected. In the first case, the helper function raises an interrupt with the vector number in the guest code, through setting of the corresponding data structures in QEMU. Then it calls `longjmp()` to jump to the latest context saved by `setjmp()`, so this function never returns. When executing line 2 following ③, the condition is not satisfied because `setjmp()` returns the argument `1` of `longjmp()`. Therefore, lines 3 – 4 will not be executed and the interrupt will not be repeatedly handled, which achieves the exact interrupt semantics. When the execution runs into the next round of the outside `for` loop, this pending interrupt will be handled in `do_interrupt()`.

6.2 Pill Hiding

Our proposed pill hiding strategy goes through three main stages: 1) detect pill instructions in the guest; 2) freeze the guest after the corresponding host code for the guest instruction has been executed; and 3) overwrite register and memory values using correct information learned from physical machines.

To detect pills, we need to compare the guest code with known pill instructions in run time. This can be achieved using either mnemonics or opcode. We choose the first approach since QEMU has a built-in disassembler.

To freeze the guest at the right point, we need to build a communication mechanism between QEMU and the guest. Debuggers achieve a similar functionality by replacing user-defined breakpoints with interrupt instructions. We cannot apply the same approach by inserting interrupts into translated code, since it will cause a trap between QEMU and the host instead. Actually, this is the reason why TCG needs to replace the guest interrupts with a call to the helper function as discussed in the previous subsection. To address this problem, we modify the QEMU's translation mechanism and utilize its interrupt handling mechanism as shown in Figure 5.

We monitor each guest instruction at line 1 by disassembling the current instruction in `pc_ptr`. If this instruction is not a pill, we directly translate it at line 11. If it is a pill, we check if the state before this instruction is saved. If not, this is the first time we encounter this instruction and we generate a `0x20` interrupt, otherwise we generate a `0x21` interrupt. Neither of these interrupt values are used by Windows. Generation of an interrupt calls `helper_raise_int()` in Figure 4 which brings the control to `do_interrupt()` as it does for other interrupt vectors. In this function we add new interrupt handlers for `0x20` and `0x21` interrupts. The handler for `0x20` saves the system state. The handler for `0x21` applies the hiding rules by overwriting the registers and memory with the values that a physical machine would set. The hiding rules can be devised by grouping pill instructions based on the resource that is the symptom of the pill (it is different in the physical and the virtual machine) and input parameter ranges. For example, we find 61 FPU instructions that always raise exceptions different from Oracles if their operands are in specific value ranges. When we detect these instructions and their operands fall in these specific ranges, we can raise the exceptions that occur in the Oracles. This would handle around 1,500 pills. Thus we can hide the presence of the pills without reimplementing instruction semantics. We emphasize here that only pills whose symptoms are not kernel registers can be hidden by our approach.

```
 1 curr_insn = disas(pc_ptr)
 2 if (curr_insn is pill)
 3     if (saved == false)
 4         gen_int(0x20)  // save states
 5         saved = true
 6     else
 7         pc_ptr = trans(pc_ptr)
 8         gen_int(0x21)  // apply hiding rules
 9         saved = false
10 else
11     pc_ptr = trans(pc_ptr)
```

Figure 5: Hooking on QEMU Translation

7 Conclusion

Virtualization is crucial for malware analysis, both for functionality and for safety. Contemporary malware tests 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.

In this paper we propose Cardinal Pill Testing that requires moderate manual action to identify ranges for input parameters for each instruction in a CPU manual, but then automatically that devises tests to enumerate the differences between a physical and a virtual machine. 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 user-space instructions that affect defined resources, Cardinal Pill testing identifies all test pills that

could be used to generate all possible individual pills. 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. Our future work will focus on this direction. Yet other pills we have discovered stem from instructions that modify kernel-level resources. We do not properly test the initialization of these instructions because that would require reboot of machines and would be too time-consuming. Thus, we cannot claim completeness for pills that relate to kernel-level resources. We plan to test these extensively in our future work.

Acknowledgments

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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security for the Space and Naval Warfare Systems Center, San Diego.

References

[1] BAJCSY, R., BENZEL, T., BISHOP, ET AL. Cyber Defense Technology Networking and Evaluation. *Commun. ACM 47*, 3 (2004).

[2] BALZAROTTI, D., COVA, M., KARLBERGER, C., ET AL. Efficient Detection of Split Personalities in Malware. In *Network and Distributed System Security (NDSS)* (2010).

[3] BARFORD, P., AND BLODGETT, M. Toward Botnet Mesocosms. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).

[4] BAYER, U., KRUEGEL, C., AND KIRDA, E. TTAnalyze: A Tool for Analyzing Malware. In *European Institute for Computer Antivirus Research (EICAR) Annual Conference* (2006).

[5] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC* (2005).

[6] BitBlaze: Binary Analysis for Computer Security. `http://bitblaze.cs.berkeley.edu/`.

[7] BRANCO, R. R., BARBOSA, G. N., AND NETO, P. D. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat* (2012).

[8] CHEN, X., ANDERSEN, J., MAO, Z., 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).

[9] DINABURG, A., ROYAL, P., ET AL. Ether: Malware Analysis via Hardware virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)* (2008).

[10] FERRIE, P. Anti-Unpacker Tricks. `http://vpn23.homelinux.org/Anti-Unpackers.pdf`.

[11] FERRIE, P. Attacks on Virtual Machine Emulators. *Symantec Security Response* (2006).

[12] GOOGLE. Android Emulator. `http://developer.android.com/tools/devices/emulator.html`.

[13] INTEL. Intel 64 and IA-32 Architectures Software Developers Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[14] JOHN, J. P., MOSHCHUK, A., GRIBBLE, S. D., ET AL. Studying Spamming Botnets Using Botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[15] KANG, M. G., YIN, H., HANNA, S., ET AL. Emulating Emulation-resistant Malware. In *Proceedings of the First ACM Workshop on Virtual Machine Security (VMSec)* (2009).

[16] KREIBICH, C., WEAVER, N., ET AL. GQ: Practical Containment for Measuring Modern Malware Systems. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC)* (2011).

[17] LAWTON, K. P. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 29es (1996).

[18] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)* (2011).

[19] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. 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.

[20] MARTIGNONI, L., PALEARI, R., FRESI ROGLIA, G., ET AL. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)* (2010).

[21] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., ET AL. Testing CPU Emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)* (2009).

[22] PÉK, G., BENCSÁTH, B., AND BUTTYÁN, L. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proceedings of the Fourth European Workshop on System Security (EuroSec)* (2011).

[23] SONG, C., ROYAL, P., AND LEE, W. Impeding Automated Malware Analysis with Environment-Sensitive Malware. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec)* (2012).

[24] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. 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).

[25] SUN, M.-K., LIN, M.-J., CHANG, M., ET AL. Malware Virtualization-Resistant Behavior Detection. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)* (2011).

[26] YAN, L.-K., JAYACHANDRA, M., ZHANG, M., 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).