Testing System Virtual Machines

Lorenzo Martignoni† Roberto Paleari‡ Giampaolo Fresi Roglia‡ Danilo Bruschi‡

Dipartimento di Fisica†
Università degli Studi di Udine
Udine, Italy
lorenzo.martignoni@uniud.it

Dipartimento di Informatica e Comunicazione‡
Università degli Studi di Milano
Milano, Italy
{roberto,gianz,bruschi}@security.dico.unimi.it

ABSTRACT

Virtual machines offer the ability to partition the resources of a physical system and to create isolated execution environments. The development of virtual machines is a very challenging task. This is particularly true for system virtual machines, since they run an operating system and must replicate in every detail the incredibly complex environment it requires. Nowadays, system virtual machines are the key component of many critical architectures. However, only little effort has been invested to test if the environment they provide is semantically equivalent to the environment found on real machines. In this paper we present a methodology specific for testing system virtual machines. This methodology is based on protocol-specific fuzzing and differential analysis, and consists in forcing a virtual machine and the corresponding physical machine to execute specially crafted snippets of user- and system-mode code and in comparing their behaviors. We have developed a prototype, codenamed KEmuFuzzer, that implements our methodology for the Intel x86 architecture and used it to test four state-of-the-art virtual machines: BOCHS, QEMU, VirtualBox and VMware. We discovered defects in all of them.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Software testing, fuzzing, virtualization, emulation, automatic test generation

1. INTRODUCTION

Virtual machine technologies have been widely and successfully used for the last 30 years in many contexts [32]. Practically speaking, a virtual machine is an isolated environment that executes software in the same way as the physical system for which the software was developed. Virtual machines can be classified in two main classes, according to the type of software they execute: *process virtual machines* execute an individual process, while *system virtual machines* execute full operating systems. Typical usages of process virtual machines are cross-platform portability, profiling, and dynamic binary optimization. On the other hand, typical usages of system virtual machines are resources consolidation [1], applications provisioning, simplification of maintenance, system integration [20], development [10], and security [2].

Virtual machines are very complex pieces of software. This is particularly true for system virtual machines, since they have to offer an execution environment suitable for running a commodity *guest* operating system and its applications. Thus, the most important requirement for a system virtual machine is to replicate in every detail the execution environment found on physical machines. From a theoretical point of view, nothing prevents to develop a virtual machine that satisfies this requirement. However, the development of this class of software is very challenging because of the complexity of modern computer architectures and because efficiency has become the main priority. Many researchers have invested a lot of efforts in the development of new techniques for building efficient system virtual machines. Traditionally, system virtual machines were implemented using software emulators that emulated the CPU and I/O peripherals. Although this approach is still in use for certain applications, modern virtual machines improve efficiency by executing natively on the physical CPU part of the code of the guest. Recently, hardware vendors have started to extend their architectures to introduce new capabilities to facilitate virtualization and to maximize the amount of guest code that can be run natively [22]. Unfortunately only little effort has been invested in developing specific testing methodologies for this class of software. Since system virtual machines are employed in a variety of critical applications and since bugs might have very dangerous implications [35], their thorough testing must be taken very seriously.

In this paper we propose a testing methodology specific for system virtual machines. We extend our previous work on this topic [18], and present a much more powerful methodology that is independent from the technique used by virtual machines to execute guest code. More precisely, in the past our testing technique was specific for testing CPU emulators (i.e., virtual machines based on CPU emulation) and limited to the testing of emulated user-mode code. The technique we propose in this paper can be used to test both user-

and system-mode code and thus can also be applied to CPU virtualizers (i.e., virtual machines that rely on native code execution). The proposed methodology is based on the assumption that the CPU of a perfect virtual machine behaves exactly as the physical CPU it simulates. Therefore, the intent of the testing is to verify whether such assumption holds. More precisely, our testing technique allows to discover sequences of instructions that, when executed, cause the CPUs of the virtual and of the physical machines to behave differently. The methodology is based on protocol-specific fuzzing [33] and differential analysis [19]. We use fuzzing to generate automatically input states for the testing, and differential analysis to detect anomalous behaviors. Since the expected behavior corresponds to that we observe in the CPU of the physical machine, any difference between the behaviors of the physical and virtual machines is clearly anomalous, and consequenlty a symptom of a defect in the latter.

We implemented a prototype, codenamed KEmuFuzzer, for testing virtual machines for the Intel x86 architecture and we used it to test four popular system virtual machines: BOCHS [16], QEMU [3], VMWare [34], and VirtualBox [23]. The first two are based on emulation, while the others are based on direct native execution. In all the virtual machines we tested we found defects that lead to the corruption of the state of the guest operating system or of its applications. The experimental evaluation testified the effectiveness of our methodology and justified the significant effort we made to extend it to system-mode code and CPU virtualizers. In conclusion, we believe KEmuFuzzer should become an integral part of the development cycle of system virtual machines (e.g., for automatic regression testing).

To summarize, the paper makes the following contributions.

- We propose a fully automated testing methodology, which is based on fuzzing and differential analysis, specific for system virtual machines.

- We describe a prototype implementation for the Intel x86 architecture. The prototype is available as an open-source package at `http://security.dico.unimi.it/kemufuzzer/`.

- We present an extensive testing of four popular virtual machines. Our experimental evaluation witnessed that none of them is free from defects.

2. OVERVIEW

This section describes the approaches used in system virtual machines to simulate the physical CPU, introduces a property of virtual machines we called *transparency to guests*, and illustrates how this property can be used for testing virtual machines.

2.1 CPU Emulation and Virtualization

The processor of a physical machine, the *host*, can be programmed to run multiple *guests* and to give them the illusion that they have dedicated and complete accesses to the processor. This illusion can be realized in two ways: through CPU emulation and through CPU virtualization.

A *CPU emulator* is software that simulates the execution environment offered by the physical CPU, by emulating all the instructions. Instructions are either emulated using interpretation or just-in-time translation. They mimic in every detail the behavior of instructions executed directly by the physical CPU, with the exception that the former operate on the resources of the emulated execution environment, while the latter operate directly on physical resources. The emulated execution environment consist of: an address space (the memory), general purpose registers and other classes of registers (e.g., FPU and management registers).

CPU virtualizers can be viewed as very efficient emulators. Indeed, when the instruction set of the guest is identical to the instruction set of the host, most of the code of the guest can be executed directly as-is on the host[1]. The trick used to natively run guest code efficiently is to execute both user and system code of the guest on the host, in user-mode. Emulation is then used to execute only instructions of the system code that cannot be executed natively on the host. The complexity and efficiency of the virtual machine depends on the number of instructions that require emulation and on the complexity of discovering them. Popek and Goldberg formally described the characteristics an instruction set architecture (ISA) must meet to be easily and efficiently virtualized [26]. They identified two special classes of instructions, privileged and sensitive, and derived a sufficient condition under which an ISA can be efficiently (and easily) virtualized. A *privileged instruction* is an instruction that traps (i.e., it raises an exception) when executed in user-mode and does not trap when executed in system-mode. A *sensitive instruction* is instead an instruction that either changes the configuration of the resources in the system or whose behavior depends on the configuration of the resources. The ISA is efficiently virtualizable if the set of sensitive instructions is a subset of privileged ones. When such a condition is met, all sensitive instructions that require emulation trap naturally, since they are privileged but are executed in user mode. Thus, the host can intercept them with no effort. Since the number of sensitive instructions is typically small, the complexity of the CPU emulator needed on the host to handle these instructions is much smaller than the complexity of a traditional CPU emulator that must support the whole instruction set.

Unfortunately, the majority of ISAs do not meet the Popek and Goldberg requirement for efficient virtualization. An example is the Intel x86 ISA [28][2]. In order to allow virtualization in such an architecture all system code of the guest must be analyzed to detect *critical instructions*, that is, sensitive, but not privileged, instructions. Any block of code containing a critical instruction must then be either emulated or patched (to allow the host to intercept the critical instruction). Thus, the identification and handling of critical instructions requires a complex software component that resembles, in terms of characteristics, complexity, and proneness to defects, a traditional CPU emulator.

[1]We are (ab)using the term "CPU virtualization" to explicitly refer to the virtualization technique also known as *direct native execution* [32].

[2]Recently the Intel x86 ISA has been extended introducing hardware support for virtualization (VT-x). However, in this paper we are concerned with the testing of virtualization techniques that do not leverage such a support.

2.2 Modeling the Behavior of the CPU

In order to present clearly the problem we are addressing it is important to model precisely the way a CPU behaves. The model we use was introduced in our earlier paper [18]. In the following paragraphs we present a simplified and revisited model that accounts for CPU virtualizers.

Given a physical CPU $\mathcal{C}_P$, we denote with $\mathcal{C}_E$ a software CPU emulator that emulates $\mathcal{C}_P$. Similarly, we denote with $\mathcal{C}_V$ a CPU virtualizer that simulates $\mathcal{C}_P$ by executing some code directly on the physical CPU, and by relying on an emulator to execute privileged instructions. We use $\mathcal{C}_{VE}$ to denote the emulation module embedded in the virtualizer. Recall that on architectures that do not meet the Popek and Goldberg requirement for efficient virtualization, $\mathcal{C}_V$ has to rely on emulation, or scanning and patching, to identify and handle critical instructions. To keep our model simple, we consider the scanning and patching approach as a form of emulation.

We can imagine the CPU as an abstract machine $(\mathcal{S}, \delta)$. The set of states $\mathcal{S}$ represents all the possible configurations of the abstract machine. Briefly, a state, or a configuration, of the machine consists in the configuration of the CPU registers (general purpose registers, control registers, and FPU registers) and of the physical memory. The state-transition function $\delta \colon \mathcal{S} \to \mathcal{S}$ maps a machine state s into a new state s' by executing the instruction pointed to by the program counter. Given a state s, the resulting state $\delta(s) = s'$ depends on s and on the semantics of the instruction executed. The execution of a sequence of instructions can be described as the transitive closure of the state-transition function, $\delta^*(s) = s'$, where s' is the state reached after all instructions have been executed.

We denote the state-transition functions of $\mathcal{C}_P, \mathcal{C}_E$, and $\mathcal{C}_V$ respectively as $\delta_{\mathcal{C}_P}$, $\delta_{\mathcal{C}_E}$, and $\delta_{\mathcal{C}_V}$. The semantics of $\delta_{\mathcal{C}_P}$ is defined by the specification of the CPU, while the semantics of $\delta_{\mathcal{C}_E}$ depends on how the emulator is implemented and how it adheres to the specification of the CPU. The semantics of $\delta_{\mathcal{C}_V}$ instead is hybrid and can be defined as follows:

$$\delta_{\mathcal{C}_V}(s) = \begin{cases} \delta_{\mathcal{C}_P}(s) & \text{for native execution,} \\ \delta_{\mathcal{C}_{VE}}(s) & \text{for emulated execution.} \end{cases}$$

In other words, for a subset of $\mathcal{S}$, the semantics of the state-transition function of the virtualizer corresponds to that of the physical CPU. For the remaining subset of states, the semantics corresponds to that of the emulator embedded in the virtualizer to handle states that do not allow native execution. Note that, in the case of the execution of multiple instructions, the final state $\delta^*_{\mathcal{C}_V}(s) = s'$ might be obtained by switching back and forth from native to emulated execution (e.g., $\delta^*_{\mathcal{C}_V}(s) = \delta_{\mathcal{C}_P}(\ldots(\delta_{\mathcal{C}_{VE}}(\delta_{\mathcal{C}_P}(s)))))$.

This simple model of the behavior of the CPU might appear incompatible with the way traditional interrupt-driven architectures operate. However, when the CPU disables or ignores interrupts, the transition from one state to another is totally deterministic.

2.3 Transparency of Virtual Machines

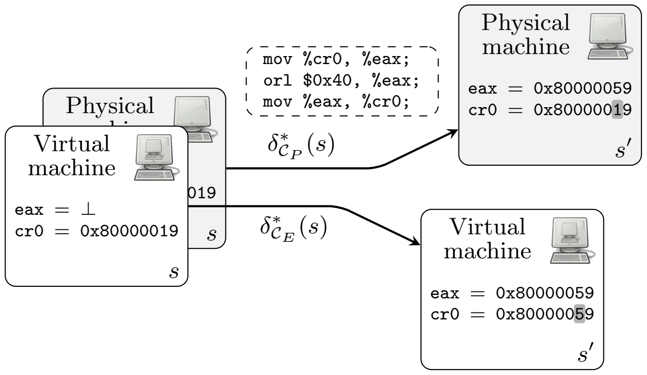
The ideal CPU emulator and the ideal CPU virtualizer behave exactly as the physical CPU. Therefore, any program should produce the same output when executed on the physical CPU and when executed in a virtual machine. Our goal is to analyze system virtual machines to tell how close they resemble the ideal one. To do that we define a property called *transparency to guests*. Transparency to guests means that guests must not be able tell if they are executed in a virtual machine or not. In other words, $\mathcal{C}_E$ is transparent if the state-transition function $\delta_{\mathcal{C}_E}$ that models $\mathcal{C}_E$ is semantically equivalent to the function $\delta_{\mathcal{C}_P}$ that models $\mathcal{C}_P$. That is, for each possible state $s \in \mathcal{S}$, $\delta_{\mathcal{C}_P}$ and $\delta_{\mathcal{C}_E}$ always transition into the same state. More formally, $\mathcal{C}_E$ satisfies the transparency property iff:

$$\forall s \in \mathcal{S} : \delta_{\mathcal{C}_E}(s) = \delta_{\mathcal{C}_P}(s).$$

In the same way we define the transparency property for $\mathcal{C}_V$; the only difference with the previous case is that transparency is implicitly assured when guest code is executed directly on the host. Therefore, $\mathcal{C}_V$ is transparent if:

$$\forall s \in \mathcal{S} \text{ that requires emulation} : \delta_{\mathcal{C}_{VE}}(s) = \delta_{\mathcal{C}_P}(s).$$

Note that our definition is restricted to the transparency with respect to the CPU and does not take into account differences in the states that might be caused by simulated I/O devices, since I/O devices are not part of the CPU.



```
mov %cr0, %eax;
orl $0x40, %eax;
mov %eax, %cr0;
```

Figure 1: Example of lack of transparency

Clearly, transparency is strictly related to correctness and transparency implies the absence of defects. Figure 1 shows a practical example of the lack of transparency, based on a real defect we found in QEMU. The code fragment sets a reserved bit (the sixth) of the control register `cr0`. The Intel x86 reference manual states that reserved bits in `cr0` preserve their value when modified [12]. As highlighted in the figure, the CPU of the virtual machine does not completely adhere to the specification. Our goal is to use the transparency property just defined to analyze CPU emulators and virtualizers to find defects in their implementation. It is worth noting that in the case of a traditional emulator, the causes of lack of transparency are imputable totally to software defects. On the other hand, in the case of a CPU virtualizer, lack of transparency could also be caused by intentional design and implementation decisions, made to facilitate virtualization. For example, a CPU virtualizer could make assumptions about the internals of the guest, or could force the guest into certain configurations that allow to minimize the complexity of the virtualizer and to minimize performance overhead. In any case, the lack of transparency can produce unexpected behaviors in guests.

2.4 Testing Transparency of Virtual Machines

CPU emulators and CPU virtualizers, on virtualization unfriendly ISAs (i.e., ISAs with critical instructions), are very complex pieces of software. Consequently, it is not that easy to guarantee complete transparency. Indeed, our experience has taught us that even CPU virtualizers on virtualization friendly ISAs sometimes fail to satisfy this property. Thus, we propose a methodology to test automatically whether such a property is satisfied or not.

Our approach to test if an emulator $\mathcal{C}_E$, for the CPU $\mathcal{C}_P$, is transparent or not is based on fuzzing [21] and differential analysis [19]. We use fuzzing to generate an input state s. Since the state space $\mathcal{S}$ is prohibitively large and since many states are equivalent for the purpose of testing, the fuzzing is protocol-specific [33]: we start from a small set of meaningful states, and we mutate them to generate new ones that try to exercise the largest class of corner-case behaviors of the CPU. Compared to traditional fuzzing, the protocol-specific approach we are using allows to concentrate the efforts mostly on meaningful states.

Given an input state s, we initialize both $\mathcal{C}_E$ and $\mathcal{C}_P$ to s. We resume the execution of the two CPUs and wait until they halt. The CPUs start with executing the instruction pointed to by the program counter (as defined in s) and terminate the execution when an exception occurs or when a special halt instruction is reached. Termination is guaranteed by s, since it is generated as a mutation of another state that guarantees termination, using a transformation that preserves this property. When both CPUs have terminated the execution we compare the final states. If no difference is found, $\delta^*_{\mathcal{C}_P}(s) = \delta^*_{\mathcal{C}_E}(s)$ holds and $\mathcal{C}_E$ is transparent with respect to s. Note that we consider δ^* instead of δ as, depending on the input state, the testing might involve the execution of more than one instruction. We use the exact same approach to test a CPU virtualizer $\mathcal{C}_V$, but we focus the efforts only on the subset of states that require emulation.

The problem we address in this paper is by far more challenging than the one we considered in our earlier work. In fact, in the past we focused the testing only on the behavior of CPU emulators in user-mode. In this paper we are extending the testing to system-mode as well. As discussed in the next section, this type of testing is much more complicated from a practical point of view.

3. KEMUFUZZER

To implement the testing methodology presented in the previous section, three major challenges must be addressed. First, in order to test the transparency of a virtual machine, we need to execute some code in the virtual and in the physical system, and then compare the state resulting from the execution. Unfortunately, since our approach is based on fuzzing and since we want to test transparency in both user- and system-mode, we might lead the physical machine into an unusable state, from which it would be impossible to regain the control without a reboot. To be able to inspect the state of the machine at any time, we need to hold complete control of the machine, even when fatal exceptions occur. Second, in our previous work, we had access only to user-mode resources and thus we assumed that the behavior of an instruction depended only on the value of its operands. By taking into account also system-mode resources, this assumption does not hold anymore. For example, the behavior of an instruction that accesses the memory now also depends on the configuration of paging, segmentation, and protection rings. As the number of configurations that affect the behavior of an instruction is significantly larger, a new technique for test-cases generation is necessary. Third, our definition of transparency assumes that the state-transition function that models the behavior of the CPU is deterministic. Nevertheless, asynchronous events (e.g., interrupts) make the execution on real CPUs non-deterministic. Consequently, we have to setup a proper execution environment that guarantees deterministic execution in all possible CPU modes. In other words, all the effect of asynchronous events must be nullified.

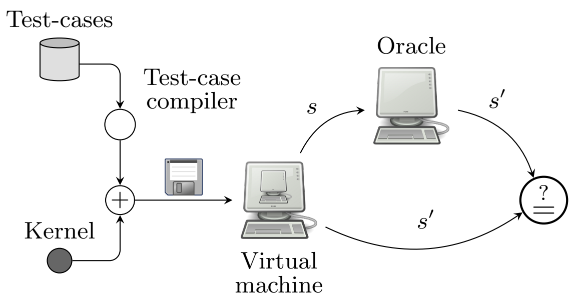


Figure 2: Overview of KEmuFuzzer

The details of the implementation we are going to present are specific for Intel x86 and for testing system virtual machines. However, the implementation can be adapted to test virtual machines for other architectures and process virtual machines as well (e.g., by recreating the environment of the host in which the process is run).

3.1 Architecture and Methodology

Figure 2 depicts the architecture of KEmuFuzzer, the system we developed for testing the transparency of CPU emulators and virtualizers for the Intel x86 architecture. KEmuFuzzer is composed of the following modules: (I) a compiler to generate test-cases from manually written templates; (II) a kernel to bootstrap the CPU of the virtual machine and to execute a test-case; (III) an oracle that executes a test-case on the physical CPU; (IV) a coordinator (not shown in the figure) to automate the process of validating the transparency of the virtual machine for a given test-case.

Given a test-case template, we use the compiler to translate the template into a stream of machine instructions. For each compiled test-case, we generate a floppy image containing a boot-loader, the kernel, and the compiled test-case. The floppy image is bootable and can be used to boot any Intel x86 compatible machine, including CPU emulators and CPU virtualizers for this architecture. We use this floppy image to boot the virtual machine under testing. The boot loader executes the kernel and the kernel initializes the environment for executing the test-case. When the environment is fully initialized, we take a snapshot of the state of the virtual machine (s). The snapshot includes the content of all the registers of the CPU and the content of the physical memory. Subsequently, we start the execution of the test-case and wait until it terminates, or a timeout occurs. At this point we take a new snapshot of the state of the virtual machine (s'_E). We start the oracle and force its initial state to s. Thus, the oracle executes immediately the test-case, in the same exact configuration in which we previously executed the test-case in the virtual machine. When the execution terminates we take a snapshot of the state of the machine (s'_P). Finally, we compare s'_E with s'_P. Any difference is a symptom that the virtual machine is not transparent and consequently buggy.

3.2 Test-cases

Test-case generation is a key issue. The state space is prohibitively large; it is essential to select test-cases that are able to exercise the largest class of behaviors of the CPU and thus to increase the completeness of the testing. Moreover, when testing CPU virtualizers, test-cases must be generated taking into account the fact that emulation is used only for certain machine states and that it would be completely worthless to test the behavior of the virtualizer with test-cases that are executed natively on the physical CPU. However, it is very difficult to predict precisely which states are handled using emulation and which are not, since that highly depends on the implementation of the virtual machine. The approach we use to generate states for the testing is based on protocol-specific fuzzing: we start with a state we believe significant for the testing and then we generate automatically new states by varying some parameters of the initial state.

A test-case consists of a sequence of one or more instructions executed starting from a well defined state s. A test-case can contain up to four blocks of instructions, each of which is executed in a different privilege level, or ring. Thus, a block of instructions can be executed in system-mode (ring 0), in user-mode (ring 3), or in any of the remaining two intermediate rings available on the Intel x86 architecture. The test-case additionally defines which of these four blocks will be executed as first by the kernel. If necessary, a block can include instructions to switch to another ring and to execute the instructions of the corresponding block.

Test-cases are not written manually but generated automatically from *templates*. Templates are manually written in assembly but can contain symbolic operators that refer to symbols of the kernel or to generator functions that return a set of concrete values. In our test-case templates, symbolic operators are written in uppercase and prefixed with the keyword `KEF_`. Table 1 briefly summarizes some of the operators we currently support. Templates are compiled with a special compiler we developed. The compiler pre-processes the assembly code to replace symbolic operators with concrete ones and then assembles the result (using the GNU Assembler [8]). When test-cases are compiled, the code located in each execution ring is always extended to include an instruction to invoke a particular software interrupt. As discussed in Section 3.3, this instruction is used to notify that the execution of the test-case has been completed without any exception. A single template can be compiled into multiple test-cases, each of which differs in the concrete values returned by the generator functions. Thus, using templates and generator functions we can automatically test the behavior of the CPU in many different situations.

Figure 3 presents three sample test-case templates. Strictly speaking, each template is an XML document. XML has been chosen to ease the writing. The root of the document has a child node for each of the four privilege levels

Symbolic operator	Description
`KEF_INTEGER(`n`)`	Generate a set of n-bit integers
`KEF_ITERATE(`$\mathrm{i}_1,\ldots,\mathrm{i}_n$`)`	Iterate over $\mathrm{i}_1,\ldots,\mathrm{i}_n$
`KEF_BITMASK(`n`)`	Generate a n-bit bitmask
`KEF_PREFIX`	Generate different combinations of certain instruction prefixes
`KEF_RAND_STR(`n`)`	Generate n random strings
`KEF_JUMP_RING(`n`)`	Switch to ring n
`KEF_PT_BASE`	Page table base address
`KEF_RING_BASE(`n`)`	Base address of ring n
`KEF_RING_CS(`n`)`	CS selector for ring n

Table 1: Examples of symbolic operators used in test-case templates

supported by the architecture. However, child nodes with no code can be omitted. An attribute of the root node (`start_ring`) controls in which privilege level the execution begins. Figure 3(a) shows a template of a test-case to test whether the emulated CPU is correctly decoding the `sysenter` instruction and correctly interpreting its semantics. The `KEF_PREFIX` symbolic operator refers to a generator function that returns different combination of instruction prefixes (e.g., `rep`, `lock`). The template is compiled into multiple test-cases, each of which obfuscates the `sysenter` instruction by prepending a different combination of prefixes. Indeed, according to the Intel x86 specification, certain combinations of prefixes and `sysenter` are not allowed (e.g., `lock sysenter`). Since the `sysenter` instruction is a user-mode instruction, but its successful execution depends on the existence of the appropriate syscall handler, we start with executing system-mode (ring 0) code to register the handler, and then transfer the execution to user-mode (ring 3). The `KEF_JUMP_RING(3)` symbolic operator is replaced with a code snippet that transfers the execution to user-mode. Figure 3(b) shows a template of a test-case to corrupt the page table and to observe how the CPU behaves in such a situation. The `KEF_INTEGER(`n`)` symbolic operator refers to a generator function that returns different n-bit integer values. Hence, each test-case generated starting from this template will corrupt the page table in a different way. The symbolic operators `KEF_RING_BASE(3)` and `KEF_PT_BASE` instead refer respectively to the base address of the user-mode data segment and the base address of the page table. The use of such operators renders templates completely independent from changes in the KEmu-Fuzzer's kernel. Finally, Figure 3(c) shows a template of a test-case used to check whether the emulated CPU properly enforces alignment checking. Execution starts in system-mode, to enable hardware-assisted alignment checks, and then switches to user-mode to perform different types of non-aligned memory accesses. The `KEF_ITERATE` symbolic operator generates test-cases that perform different types of memory accesses (by jumping to different instructions). Since Intel x86 specification says that alignment checking should be enforced only in ring 3 and only for a subset of the instructions of the instruction set, the intent of the test-case is to test whether the emulated CPU enforces alignment checking correctly (i.e., for the correct set of instructions and only in user-mode). To this aim, we use the `randomize_ring` attribute of the root node to specify that an execution ring must be randomized (i.e., different test-cases are generated,

```
<testcase start_ring="0">
  <ring0>
    // Register syscall handler
    ...
    // Jump to ring3 code
    KEF_JUMP_RING(3);
  </ring0>

  <ring3>
    // Invoke syscall
    mov $0x25, %eax;
    KEF_PREFIX sysenter;
  </ring3>
</testcase>
```

(a)

```
<testcase start_ring="0">
  <ring0>
   // Load flat data segment
   ...
   // Calculate PTE address
   movl KEF_RING_BASE(3), %eax;
   ...do some math on %eax...
   addl KEF_PT_BASE, %eax;
   // Flip some bits in the PTE
   movl (%eax), %ebx;
   btc KEF_INTEGER(5), %ebx;
   movl %ebx, (%eax);
   ...
   // Jump to ring3 code
   KEF_JUMP_RING(3);
  </ring0>

  <ring3>
   movb $0x0, 0x0;
  </ring3>
</testcase>
```

(b)

```
<testcase randomize_ring="3" start_ring="0">
 <ring0>
   // Set AM bit in CR0
   mov %cr0, %eax;
   orl $0x40000, %eax;
   mov %eax, %cr0;
   // Jump to ring3 code
   KEF_JUMP_RING(3);
 </ring0>

 <ring3>
   // Enable AC bit in EFLAGS
   ...
   // Perform unaligned memory acceses
   jmp KEF_ITERATE(lab1, lab2, ...);
   lab1: KEF_PREFIX movb $0x0, 0x1;
   lab2: KEF_PREFIX fld 0x423;
   ...
 </ring3>
</testcase>
```

(c)

Figure 3: Sample test-case templates: (a) `sysenter` with different prefixes; (b) fuzzing of a page table entry; (c) non-aligned memory access with alignment checks enabled.

each of which executes the current ring 3 code in a different ring). In summary, this sample template will be compiled in multiple test-cases: some of them will perform different types of non-aligned memory accesses in ring 3, some others in 1, and so on.

It is worth pointing out that KEmuFuzzer's kernel initializes the environment always in the same way. Although all test-cases are executed starting from the same initial state s, it is theoretically feasible to test the behavior of the CPU in any possible state. Indeed, it is possible to set the CPU in the desired state by embedding the appropriate code directly into the test-case. That is exactly what happens with the three test-case templates of Figure 3. The first one registers a custom syscall handler, the second one alters the page table entry that corresponds to the base address of the user-mode data segment, and the third one enables alignment checking.

3.3 Kernel

KEmuFuzzer uses a custom kernel we developed, to bootstrap the environment in which test-cases are executed. More precisely, the kernel is responsible for initializing the CPU in the execution mode target of the testing (e.g., 32-bit protected mode with paging enabled), for starting the execution of the test-case, and for notifying the virtual machine when bootstrap is completed and when the execution of the test-case is terminated. The kernel is optimized to minimize bootstrap time and to minimize memory usage. Currently our kernel boots in less than half a second and requires less than 4Mb of physical memory to run.

The kernel communicates with the virtual machine and with the oracle through a specific hardware I/O port; we call this port the *notification port*. We use this trick to request the virtual machine and the oracle to dump the current machine state to a file. Further details about how this is done are given in Section 3.4 and 3.5.

Figure 4 shows the memory layout at the end of bootstrap and just before the execution of a test-case. The following details about the kernel are specific for initializing the Intel x86 CPU in 32-bit protected mode with paging enabled. However, with minimal modifications the kernel can initialize the environment in other modes of operation (e.g., virtual mode, long mode, protected mode with physical address extension). The kernel starts by enabling protected mode and configuring the Programmable Interrupt Controller (PIC). Subsequently, the kernel initializes the Global Descriptor Table (GDT). For each of the four protection rings we create a task, and for each task we create a segment, of 4Kb, to hold simultaneously the code, the data, and the stack of the task. The stack occupies the second half of the segment. The kernel uses segmentation to prevent a test-case from accidentally corrupting its main memory. The kernel subsequently configures and enables paging. The page directories and the page tables are initialized such that memory address translation is an identity function: the virtual address a is translated to the physical address a. The reason for such a configuration is to simplify the analysis of the machine state. The dumps of the state we produce include the content of the entire physical memory. Therefore, such a page mapping allows us to inspect the virtual memory with no effort. After paging has been enabled, the kernel initializes the Interrupt Descriptor Table (IDT) by registering special exception and interrupt handlers. Exception handlers write a command to the notification port to signal the occurrence of an exception and to request dump of the state; after the notification they halt the CPU. Interrupt handlers instead immediately resume normal execution, by executing the `iret` instruction. This approach guarantees a deterministic execution, because unpredictable asynchronous interrupts do not alter the state of the machine. The kernel also configures a special interrupt handler (interrupt 31) that is used to notify the end of the execution of the test-case without any exception. Like exception handlers, this interrupt handler writes a command to the notification port to request a dump of the state and then halts the CPU. This handler is invoked directly by test-cases. When test-cases are compiled, the compiler appends at the end of the sequence of instructions of each ring an extra instruction to trigger a software interrupt and invoke this special handler (see Section 3.2). After the IDT has been configured, the kernel enables interrupts. After that, the kernel writes a command to the notification port, to notify the end of the bootstrap and to request a dump of the

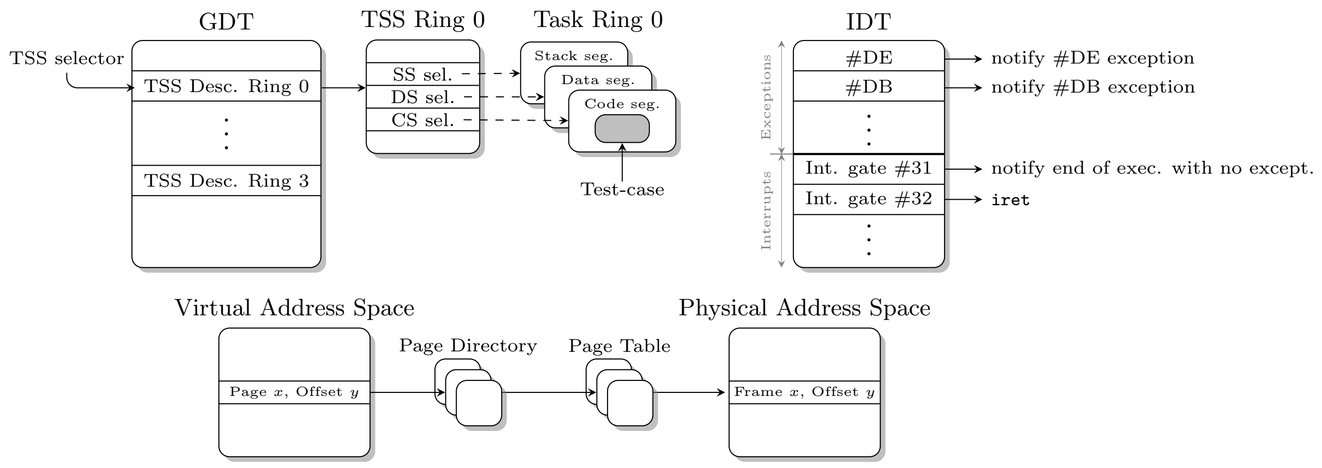


Figure 4: Memory layout of the environment used to execute test-cases

machine state. Finally, the kernel starts the execution of the test-case, through a task switch.

As mentioned before, the test-case is executed into a separate task. It is worth pointing out that, since the test-case can also contain additional initialization code, the memory layout of the environment can be further modified directly from the test-case, thus allowing to test the virtual machine in virtually any possible state. Even though segmentation is used to prevent access to sensitive memory locations from within the test-case, system-mode instructions of the test-cases can be used to remove segment limits, to grant access to all memory locations, and subsequently to configure the CPU in the desired state. Our test-case compiler provides a special symbolic operator for facilitating this operation.

The test-cases are embedded directly into the kernel: we create multiple copies of the kernel and in each copy we embed a different test-case. The kernel's executable contains four placeholder sections, corresponding to the code segments of the four rings (see Figure 4). We overwrite the content of these sections with the code of the test-case. Moreover, we patch the instruction used to start the execution of the test-case, to start executing in the desired ring.

3.4 CPU Emulators and Virtualizers

To execute a test-case in a virtual machine we simply boot it using the floppy image containing the boot loader, the kernel and the test-case. This approach works very well since all virtual machines we are aware of support booting from a floppy. When bootstrap is completed and after test-case execution terminates, the kernel alerts the virtual machine through the notification port. The virtual machine must intercept the request and dump the current state to a file.

If the source code of the virtual machine is available, the addition of such a functionality is quite simple. The approach we use, and suggest, is to register a new virtual hardware device and to associate it with the I/O port used by the kernel for notification. Typically, virtual machines provide an API to create new virtual devices. When the kernel writes a command on the notification port, the virtual machine suspends the execution of the guest and delivers the command to the device. The device has direct, or indirect, access to the internal state of the machine and can dump the state to a file. Obviously the complexity of this task varies from one virtual machine to another. For example, BOCHS offers a rich instrumentation API to intercept events (e.g., exceptions, interrupts, I/O) and to inspect the state of the guest. On the other hand, QEMU and VirtualBox do not provide an instrumentation API. Nevertheless, it is still quite easy to add a new device driver and to dump the state of the CPU. The only precaution is to ensure that the state of the guest visible from the device is in sync with the effective state.

If the source code of the virtual machine is not available, the use of the debugging interface of the virtual machine is the only viable alternative. We used this approach with VMware. Kernel notification can be detected using breakpoints, set on I/O instructions used for the notifications. The state of the guest can instead be inspected and dumped using the appropriate commands of the debugger. The drawback of this approach is that the debugging interface is typically interactive and very slow. Moreover, the debugger might not expose completely the internal state of the guest. In such a situation, the kernel must be modified to store in memory the state of the registers that are not accessible from the debugger.

3.5 Oracle

The ideal oracle is the physical CPU. We should perform the testing by either booting and running the test-case on a real machine using a floppy or the network. Unfortunately, this approach is not practical for two reasons. First, we need to dump the state s' resulting from the execution of the test-case to compare it with the state obtained in the virtual machine. Practically speaking, that means that the kernel used to bootstrap the environment and to execute the test-case must be able to interact with a device (e.g., a disk or a network card) to dump the state of the CPU. To do that, the kernel must include the appropriate device driver. Second, test-cases are specifically crafted to exercise a large class of behaviors of the CPU, including bringing it into invalid states to see how it reacts. Therefore, some test-cases might render the CPU completely unusable. For example, the processor might enter an infinite loop that prevents the kernel from regaining the control of the execution, or the kernel might get corrupted and unable to dump the state of the machine. In conclusion, the oracle based on the naïve use of the physical CPU, besides requiring a much more complex kernel, is also not suited for automatic testing with tens of thousands of test-cases: there exist certain situations in which it is not possible to dump the state of the machine or in which manual intervention is needed (e.g., to reset physically the CPU).

For these reasons, the oracle we use is based on a *hardware-assisted virtual machine*. At first sight this choice might appear to contradict our initial claims and our goal. After all, we are proposing to find bugs in a virtual machine using another type of virtual machine. In the next paragraphs we will show that this approach is instead very easy to implement, and that specific assumptions about the peculiar type of guest we need to execute allow us to develop a hardware-assisted virtual machine which is functionally equivalent to the ideal oracle previously described.

Our oracle leverages Intel technology for hardware assisted virtualization, namely VT-x [22]. By using hardware-assisted virtualization, we can observe the execution of the test-case on the physical CPU without losing the ability to interrupt the execution and to inspect the state of the CPU at any time, even when it enters invalid states. Intel VT-x technology transforms Intel x86 (and x86-64) ISA into something much more virtualization-friendly than a ISA that meets Popek and Goldberg minimal requirement for efficient virtualization. Besides not having any critical instruction, VT-x allows to configure dynamically which instructions must trap and in which conditions. Furthermore, VT-x includes a new mode of operation (VMX), that essentially adds new higher privilege rings for running the virtual machine monitor, that is, the software component the host uses to manage guests. The introduction of these new rings allows a clear separation between host and guests, which is *not invasive* for guests. Indeed, system code of the guest is executed natively on the CPU, in system-mode. Nevertheless, the host still needs to assume the control of the guest in certain situations (e.g., to redirect I/O operations to devices emulated via software). Clearly, the challenge is to develop a minimalistic software component for the host, that is sufficiently sophisticated to execute a test-case and to hold complete control of the execution, but that is also simple enough to be verifiable.

The aim of the oracle is solely to execute a particular test-case in a particular state of the CPU. In light of that, the virtual machine monitor can be drastically simplified. Instead of booting the kernel in the oracle, we can initialize manually the state of the oracle by loading s, the state of the virtual machine at the end of the bootstrap and that precedes the execution of the test-case. This approach has two major benefits. First, by initializing the state of the oracle to s we have the guarantee that the state obtained at the end of the execution of the test-case is not polluted by some differences introduced during bootstrap. Second, the bootstrap requires to execute real-mode instructions (that require emulation even with VT-x) and to communicate with I/O devices (e.g., to initialize them and to load the kernel). By avoiding to bootstrap the execution environment in the oracle, the complexity of the virtual machine monitor reduces drastically. In fact, the virtual machine monitor does not need to emulate real-mode instructions and does not need to emulate any I/O device.

Our oracle is based on a stripped down version of KVM (Kernel-based Virtual Machine) [14], a virtual machine monitor for GNU/Linux. KVM architecture is very simple because it runs only on processors with hardware support for virtualization. We have further simplified its architecture making specific assumptions about the peculiar type of guest we need to run. KVM exposes to the programmer an interface (`/dev/kvm`) to create and manage virtual machines. More precisely, this interface allows to read and write the values of all the registers of the CPU, including FPU and control registers, and to read and write the physical memory of the guest. Using this interface we can create a virtual machine that is already initialized and ready to execute a test-case. Roughly speaking, we can create a virtual machine and initialize it to the state (s) preceding the execution of the test-case. Thus, when we start the virtual machine, the test-case is immediately executed. KVM also allows to intercept and emulate I/O operations. Our stripped down version of KVM simply terminates the execution of the guest at the first I/O operation. This approach has two advantages. First, we can easily identify and ignore test-cases whose final state might be polluted by the interaction with devices external to the CPU. Second, the component for I/O emulation is by far the most complex component of the virtual machine monitor. By interrupting a test-case at the first I/O operation we have the guarantee that a potential defect in the component for I/O emulation will not influence the results of our oracle.

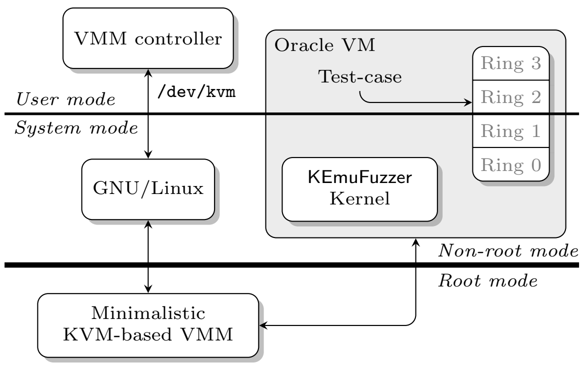


Figure 5: Oracle based on VT-x

Figure 5 shows the architecture of our oracle based on hardware-assisted virtualization. The core of the oracle is a small virtual machine monitor (VMM), based on KVM, that runs in *root-mode*, the new mode introduced in VT-x to run virtual machines without segregating guests to user-mode. This component is responsible for running and controlling the guest and can intercept certain guest's operations. As described in the previous paragraph, our virtual machine monitor minimizes the number of guest's operations that are intercepted. The oracle also consists in a user-space controller that is run on the host and that communicates with the virtual machine monitor. The controller is used to instantiate virtual machines and to dump the state of the guest. Clearly, we cannot formally prove that our oracle is equivalent to the ideal one. However, we have thoroughly inspected the code of our minimalistic virtual machine monitor, and we have verified that the defects we found during the evaluation are not imputable to defects in the oracle.

4. EVALUATION

We used KEmuFuzzer to test four virtual machines: two CPU emulators (BOCHS and QEMU) and two CPU virtualizers (VMware and VirtualBox). None of the virtual machines was found to be completely transparent to guests.

4.1 Experimental Setup

The evaluation of our testing methodology was performed using an Intel Core2 Duo (3.00GHz) machine, running Debian GNU/Linux, with kernel 2.6.31 (64-bit version). The physical processor supported the following features: MMX, SSE, SSE2, SSE3, SMX, and VT-x. We tested the following versions of the virtual machines: BOCHS CVS (7th October 2009), QEMU 0.11.0, VirtualBox OSE 3.0.8, and VMware Workstation 7.0. For the two CPU virtualizers, we disabled the support for hardware-assisted virtualization since our goal was to test traditional CPU virtualizers that do not leverage such technology.

4.2 Test-cases

We manually wrote 67 test-case templates, that were compiled into 1915 test-cases. The number of test-cases generated from each template depended on the type and number of symbolic operators used in the template. We roughly classified the test-case templates in the categories shown in Table 2. Templates in the "*privileged instructions*" category consist in privileged instructions that are executed in multiple privilege levels, usually with different combinations of prefixes. The intent of this class of test-cases was to test whether the CPU of the virtual machine implements instruction decoding and privilege checks correctly. The test-case templates belonging to the "*control registers*" category manipulate CPU control registers (e.g., `cr0` and `cr4`) to alter the execution mode of the CPU and some reserved bits. A significant percentage of the hand-written test-case templates instead belong to the "*memory management*" category. These test-cases alter the configuration of several memory-management structures (e.g., page tables, segment descriptors) to test how the CPU of the virtual machine responds to abnormal configuration of the memory management unit. The "*control transfers*" category includes test-cases that modify the execution flow through control transfer instructions or privilege switches. The "*FPU*" category encompasses test-cases that affect the operating mode of the floating-point unit (e.g., `wait` and `emms`). We intentionally did not test general-purpose floating-point instructions, as CPU virtualizers typically execute them natively. The category called "*random*" includes a template to generate sequences of random instructions and to execute them in system-mode. We also wrote other types of test-cases that do not fit any of the aforementioned categories. These test-cases are included in the category called "*others*".

We extended the set of manually-written templates with other automatically generated templates using the method, called "CPU-assisted test-case generation", we presented in our earlier paper. Briefly, this method uses directly the CPU to explore the instruction set and to identify the format of the instructions. We used this information to generate a set of test-cases that covers the large majority of the instruction set, while minimizing redundancy. Each of the templates generated with this approach executed a different instruction, in system-mode.

4.3 Experimental Results

Table 2 shows the results of our evaluation. Table 3 instead shows the average test-case execution time and the timeout on test-case execution time. For each virtual machine, Table 2 reports the number of test-case templates and the number of compiled test-cases for which we observed a non-transparent behavior. The numbers in the table witness that our initial claim, that complete transparency to guests is difficult to achieve, was correct. Indeed, no virtual machine was found to be completely transparent and thus free of defects. Moreover, as described later, the results of the evaluation show that, by extending the testing to system-mode, we are able to detect a broader class of defects that would have not been detected with sole user-mode testing. The numbers in the table also show the effectiveness of the protocol-specific fuzzing approach we used to generate test-cases. In many cases, only few of the test-cases generated from the same template triggered a defect. It is worth pointing out that the gravity of the defects we found varies from case to case. Some of the defects we found are very serious. Others instead should not negatively affect the execution of popular guests (e.g., GNU/Linux and Microsoft Windows); however, less popular guests might fail to work properly. Few differences we found are instead not directly imputable to bugs in the virtual machine, but rather to "undefined" corner-case behaviors (e.g., the Intel x86 specification does not define the value of some status flags for certain arithmetical and logical operations). Nevertheless, guests could still rely on such undefined, but deterministic, behaviors to detect if they are executed in a real or in a virtual machine [25, 29].

In the following paragraphs we briefly describe the defects we found in the tested software. In several situations we noticed non-transparent behaviors, regardless of the instructions being executed. As an example, some virtual machines never update GDT entries when accessed; others, always present some discrepancies in the state of the FPU. We decided to manually exclude these omnipresent differences from the tally to have more significant results. We are currently in touch with the developers; some of the defects we have found have already been acknowledged and patched.

4.3.1 BOCHS

The emulator is fairly perfect. Indeed, the authors stated that each release is preceded by a testing cycle using a methodology very similar to the one we proposed in our earlier paper. Nevertheless, we were able to found some unknown defects. For example, we found that the instructions used for fast system call invocation (e.g., `sysenter` and `sysexit`) corrupt an attribute (i.e., the type) of the code segment. According to the Intel x86 manual, when one of these instructions is executed, the type of the code segment should be set to "read/write" and "accessed". BOCHS erroneously marked the segment as non accessed. The defect was confirmed by the authors, and corrected in the latest versions of the emulator. The behavior of the `bswap` instruction, when it references a 16-bit register, is also non transparent. The Intel reference states the behavior of the instruction is undefined when it is executed with 16-bit operands. This is an

Category			BOCHS		QEMU		VirtualBox		VMware	
Description	*Templ.*	*Test-cases*	*Templ.*	*Test-cases*	*Templ.*	*Test-cases*	*Templ.*	*Test-cases*	*Templ.*	*Test-cases*
Privileged instructions	9	161	1	1	5	45	1	1	1	1
Control registers	7	181	1	2	6	85	5	75	1	1
Memory management	25	418	7	17	12	67	10	44	13	45
Interrupts/exceptions	9	39	2	2	7	17	5	5	0	0
Control transfer	4	45	1	6	3	35	3	35	3	35
FPU	3	3	0	0	0	0	0	0	0	0
Others	9	44	2	2	3	14	2	2	0	0
Random	1	1024	1	45	1	505	1	437	1	59
CPU-assisted	2703	2703	7	7	302	302	291	291	51	51
Total	**2770**	**4618**	**22**	**82**	**339**	**1070**	**318**	**890**	**70**	**192**

Table 2: Results of the evaluation

artifact of our testing methodology and cannot be properly considered a real defect of the emulator.

4.3.2 QEMU

We found several defects in QEMU. The most serious one causes a crash of the emulator. More precisely, we found that the emulator is not able to handle hardware breakpoints set on instructions in a memory segments with non-null base address. When such a breakpoint is hit, the emulator crashes. Another debug-related defect we found is that the emulator never sets the "resume flag" on exception. Since this flag is used to temporarily disable debug exceptions from being generated for instruction breakpoints, the emulator could enter an infinite loop. We also found that the emulator never marks GDT entries as accessed, raises the wrong exception for memory accesses beyond segment limits, and it does not support hardware-assisted alignment checking. Finally, we found that some general purpose instructions are not properly decoded and that several illegal combinations of prefixes and system-mode opcodes are considered valid and executed.

4.3.3 VirtualBox

VirtualBox handles critical instructions using both emulation and scanning and patching. Scanning and patching is used to virtualize the execution of code fragments that are frequently emulated. VirtualBox's emulation module is based on QEMU. During the testing, VirtualBox never entered native execution mode. We inspected the source code and found that native execution mode was inhibited by the configuration of our execution environment. We modified our kernel to meet this requirement and came across another and more serious problem: VirtualBox crashed on any attempt to enter native execution mode. The crash is obviously a symptom of lack of transparency. We speculate that VirtualBox makes specific assumptions about the guest and that our guest violates them. We decided to use the original kernel for the testing. In conclusion all the defects we found in VirtualBox are real and a subset of the defects we found in QEMU. However, some of them might not be reproducible with guests that do not trigger the earlier described bug.

4.3.4 VMware

Like BOCHS, VMware presented just few defects. In multiple test-cases we observed that the `ret` and `retn` instructions, used to return from function calls, are not properly emulated. Indeed, the virtual machine corrupts the state of the guest if an exception is raised during the return from a function. We also found that accessed entries of the GDT and accessed entries of the page table are not marked as such.

BOCHS[1]	QEMU[1]	VirtualBox[1]	VMware[2]	Oracle[1]
3.01s	0.74s	2.47s	25.78s	0.83s

Table 3: Average test-case execution time (timeout on test-case execution time was: 10s[1] and 40s[2])

5. RELATED WORK

Fuzz-testing has been proposed by Miller *et al.* in 1990 [21], but it is still widely used for testing different types of applications. However, pure random fuzzing cannot guarantee a reasonable code coverage in case of applications that require a particular format of the input (e.g., a XML document or a well formed Java program). For this reason, several protocol-specific fuzzing techniques have been developed that leverage domain-specific knowledge [6, 13, 33]. Another approach consists in building constraints that describe what properties are required for the input to trigger the execution of particular program paths, and then use a constraint solver to find inputs with these properties [5, 9, 30, 17].

The idea of detecting software defects by comparing the behavior of two or more software components for the same input is known as differential testing [19]. Differential testing has previously been used in a variety of contexts, including computer security [4], flash file systems [11], and grammar-driven functionality [15]. In [31] a technique based on differential analysis is used for testing Java Virtual Machines (JVMs). The idea is to feed the same test-case to different JVM implementations and to compare their output. Similarly to our test-case templates, they apply random mutators to perturb a meaningful input.

System virtual machines are widely used in computer security. One of their major applications is the dynamic analysis of malicious programs. Researchers have invested a lot of efforts to analyze state-of-the-art virtual machines to detect non-transparent behaviors that can be used to detect dynamic analysis attempts [7, 24, 27, 29]. All the non-transparent behaviors detected using our testing methodology could be used for such a purpose [25].

6. CONCLUSIONS

We presented an automated methodology for testing system virtual machines. The proposed methodology can be used to test their correctness at executing both user- and system-mode code, and applies to virtual machines based on emulation and on native execution. We implemented a prototype for the Intel x86 architecture, codenamed KEmuFuzzer, and used it to test four state-of-the-art virtual machines. We discovered multiple defects in all of them. We believe KEmuFuzzer should become an integral part of the development cycle of system virtual machines. In the future we plan to extend the testing to all possible mode of operations of the CPU (e.g., virtual 8086 mode and real mode) and to investigate the applicability of our methodology to fully-featured hardware-assisted virtual machines.

7. REFERENCES

[1] Amazon elastic compute cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Anubis. `http://anubis.iseclab.org/`.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, Berkeley, CA, USA, 2005.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Sept. 2007.

[7] P. Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec, 2006.

[8] GNU Binutils. `http://gnu.org/software/binutils/`.

[9] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[10] Google Inc. Android emulator. `http://code.google.com/android/reference/emulator.html`.

[11] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 621–631, 2007.

[12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Sept. 2009. System Programming Guide, Part 1.

[13] R. Kaksonen. A Functional Method for Assessing Protocol Implementation Security. Technical report, VTT Electronics, 2001.

[14] Kernel-based Virtual Machine (KVM). `http://linux-kvm.org/`.

[15] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *18th IFIP International Conference on Testing Communicating Systems (TestCom 2006)*, 2006.

[16] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, Sept. 1996.

[17] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[18] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*. ACM, July 2009.

[19] W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1), 1998.

[20] Windows XP Mode Homepage. `http://www.microsoft.com/windows/virtual-pc/`.

[21] B. P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12), December 1990.

[22] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, Aug. 2006.

[23] Oracle. VirtualBox. `http://virtualbox.org`.

[24] T. Ormandy. An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments. In *Proceedings of CanSecWest Applied Security Conference*, 2007.

[25] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*. ACM, Aug. 2009.

[26] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[27] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proceedings of Information Security Conference (ISC 2007)*, 2007.

[28] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th conference on USENIX Security Symposium*, 2000.

[29] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference*, 2005.

[31] E. G. Sirer and B. N. Bershad. Testing java virtual machines. In *Proceedings of the International Conference on Software Testing And Review*, nov 1999.

[32] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[33] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[34] VMware, Inc. `http://vmware.com/`.

[35] VMware security advisor. `http://www.vmware.com/security/advisories/VMSA-2009-0015.html`.