Testing CPU Emulators

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

A CPU emulator is a software that simulates a hardware CPU. Emulators are widely used by computer scientists for various kind of activities (e.g., debugging, profiling, and malware analysis). Although no theoretical limitation prevents to develop an emulator that faithfully emulates a physical CPU, writing a fully featured emulator is a very challenging and error-prone task. Modern CISC architectures have a very rich instruction set, some instructions lack proper specifications, and others may have undefined effects in corner-cases. This paper presents a testing methodology specific for CPU emulators, based on fuzzing. The emulator is "stressed" with specially crafted test-cases, to verify whether the CPU is properly emulated or not. Improper behaviours of the emulator are detected by running the same test-case concurrently on the emulated and on the physical CPUs and by comparing the state of the two after the execution. Differences in the final state testify defects in the code of the emulator. We implemented this methodology in a prototype (codenamed EmuFuzzer), analysed four state-of-the-art IA-32 emulators (QEMU, Valgrind, Pin and BOCHS), and found several defects in each of them, some of which can prevent the proper execution of programs.

Categories and Subject Descriptors

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

General Terms

Reliability, Verification

Keywords

Software testing, fuzzing, emulation, automatic test generation

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-338-9/09/07 ...$10.00.

1. INTRODUCTION

In Computer Science, the term "*emulator*" is typically used to denote a software that simulates a hardware system [15]. Different hardware systems can be simulated: a device [10], a CPU (Pin [16], and Valgrind [24]), and even an entire PC system (QEMU [2], BOCHS [14], and Simics [17]). Emulators are widely used today for many applications: development, debugging, profiling, security analyses, *etc.* For example, the NetBSD AMD64 port was initially developed using an emulator [23].

The Church-Turing thesis implies that any operating environment can be emulated within any other. Consequently, any hardware system can be emulated via software. Despite the absence of any theoretical limitation that prevents the development of a correct and complete emulator, from the practical point of view, the development of such a software is very challenging. This is particularly true for *CPU emulators*, emulators that simulate a physical CPU. Indeed, the instruction set of a modern CISC CPU is very rich and complex; therefore, the corresponding software implementation will unlikely be bug-free. Moreover, the official documentation of CPUs often lacks the description of the semantics of certain instructions in certain corner cases and sometimes contains inaccuracies (or ambiguities). Although several good tools and debugging techniques exist [22], developers of CPU emulators have no specific technique that can help them to verify whether their software emulate the CPU by following precisely the specification of the vendors. As CPU emulators are employed for a large variety of applications, defects in their code might have cascade implications. Imagine, for example, what consequences the existence of any defect in the emulator used for porting NetBSD to AMD64 would have had on the reliability of the final product.

Assuming that the physical CPU is correct by definition, the ideal CPU emulator mimics exactly the behaviour of the physical CPU it is emulating. On the contrary, the behaviour of an approximate emulator deviates, in certain situations, from the behaviour that one would have on the physical CPU. Some examples of the deviations we found in state-of-the-art emulators are reported in Table 1[1]. As an example, let us consider the instruction `add $0x1,0x0(%eax)`, which adds the immediate 0x1 to the byte pointed by the register `eax`. Assuming that the original value of the byte is `0xcf`, the execution of the instruction on the physical CPU, and on three of the tested emulators, the value of the byte is

[1] In this paper we use IA-32 assembly and we adopt the AT&T syntax.

Instruction	**IA-32**	**QEMU**	**Valgrind**	**Pin**	**BOCHS**
`lock fcos`	illegal instr.	`lock` prefix ignored	*no diff.*	*no diff.*	*no diff.*
`int1`	trap	*no diff.*	illegal instr.	*no diff.*	general protection fault
`fld1`	`fpuip` = `eip`	`fpuip` = 0	`fpuip` = 0	FPU virtualised	*no diff.*
`add $0x1,0x0(%eax)`	`0x0(%eax) = 0xd0`	`0x0(%eax) = 0xcf`	*no diff.*	*no diff.*	*no diff.*
`pop %fs`	`%esp = 0xbfdbb108`	*no diff.*	*no diff.*	`%esp = 0xbfdbb106`	*no diff.*
`pop 0xffffffff`	`%esp = 0xbffffe44`	*no diff.*	*no diff.*	*no diff.*	`%esp = 0xbffffe48`

Table 1: Examples of instructions that behave differently when executed in the physical CPU and when executed in an emulated CPU (that emulates an IA-32 CPU). For each instruction, we report the behaviour of the physical CPU and the behaviour of the emulators (differences are highlighted).

set to `0xd0`. In QEMU, instead, the value is not updated correctly for a certain encoding of the instruction. Many other examples of problematic instructions are known already [7, 25, 26, 27, 29]. Our goal is to develop an automatic technique to discover deviations between the behaviour of the emulator and of the physical CPU it is emulating, caused by defects in the emulation code. We are not interested in deviations that lead only to internal differences in the state (e.g., differences in the state of CPU caches), because these differences are not visible to the programs running inside the emulated environment. Indeed, no instruction allows to observe such internal state and consequently the execution of emulated programs cannot be influenced.

In this paper we present a fully automated testing methodology for CPU emulators, based on fuzzing [21]. The proposed methodology can be used to discover automatically configurations of the environment (i.e., state of the CPU registers and content of the memory) that cause different behaviours in the emulated and in the physical CPUs. To test an emulator we generate a large number of test-cases (i.e., configurations of the environment) and run these test-cases on both the emulated and the physical CPU. Then, we compare the configuration of the two environments at the end of the execution of each test-case; any difference is a symptom of an incorrect behaviour of the emulator. Given the unmanageable size of the test-case space, we adopt two strategies for generating test-cases: purely random test-case generation and hybrid algorithmic/random test-case generation. The latter guarantees that each instruction in the instruction set is tested at least in some selected execution contexts.

We have implemented this testing methodology in a prototype for IA-32, codenamed EmuFuzzer, and used this prototype to test four state-of-the-art emulators: BOCHS[14], QEMU[2], Pin[16], and Valgrind[24]. Although Pin and Valgrind are dynamic instrumentation tools, their internal architecture resembles, in all details, the architecture of traditional emulators and therefore they can suffer the same problems. We found several deviations in the behaviours of each of the four emulators, some of which represent serious defects that might prevent the proper execution of the emulated programs. For example, we have discovered instructions that can freeze QEMU, instructions that are not supported by Valgrind and thus generate exceptions, and instructions that are executed by Pin and BOCHS but that cause exceptions on the physical CPU. The results obtained witness the difficulty of writing a fully featured and specification-compliant CPU emulator, but also prove the effectiveness and importance of our testing methodology.

To summarise, the paper makes the following contributions:

- a fully automated testing methodology, based on fuzz-testing, specific for CPU emulators;
- an optimised algorithm for test-cases generation that systematically explores the instruction set, while minimising redundancy;
- a prototype implementation of our testing methodology for IA-32 emulators;
- an extensive testing of four IA-32 emulators that resulted in the discovery of several defects in each of them, some of which represent serious bugs.

The paper is organised as follows. Section 2 introduces formally the notion of *faithful emulation*. Section 3 describes in detail our algorithms for test-cases generation and how test-cases are run to detect if an emulator is not emulating faithfully the CPU. Section 4 evaluates our methodology by presenting the results of the testing of *four* CPU emulators. Section 5 discusses limitations and future work. Finally, Section 6 presents the related literature and Section 7 concludes.

2. OVERVIEW

This section describes how CPU emulators work, presents our notion of faithful emulation of a physical CPU, and sketches the idea behind our testing methodology.

2.1 CPU Emulators

With the term CPU emulator we refer to a software that simulates the execution environment offered by a physical CPU. The execution environment consists of: an address space (the memory), general purposes registers, other classes of registers (e.g., FPU and management registers), and optionally I/O ports. The CPU emulator emulates a program by executing each instruction in the emulated execution environment. Instructions are typically executed using either interpretation or just-in-time translation. Emulated instructions mimic in every detail the behaviour of instructions executed directly by the physical CPU, with the exception that the former operates on the resources of the emulated execution environment, while the latter operates on the resources of the physical execution environment.

The execution environment can be properly emulated even if some internal components of the physical CPU are not considered (e.g., the instruction cache): as these components are used transparently by the physical CPU, no program can access them. Similarly, emulated execution environments can contain extra, but transparent, components not found in hardware execution environments (e.g., the cache used to store translated code).

2.2 Faithful CPU Emulation

Given a physical CPU $\mathcal{C}_P$, we denote with $\mathcal{C}_E$ a software CPU emulator that emulates $\mathcal{C}_P$. Our ideal goal is to automatically analyse $\mathcal{C}_E$ to tell whether it *faithfully emulates* $\mathcal{C}_P$. In other words we would like to tell if $\mathcal{C}_E$ behaves equivalently to $\mathcal{C}_P$, in the sense that any attempt to execute a valid (or invalid) instruction results in the same behaviour in both $\mathcal{C}_P$ and $\mathcal{C}_E$.

To define how code is executed by a CPU we model the CPU with an abstract machine. A state of the abstract machine, $s \in \mathcal{S}$, consists of the program counter *pc*, the state of the CPU registers R, the state of the memory M, and the exception state E. For conciseness, we represent the state of the abstract machine with the tuple $s = (pc, R, M, E)$. The CPU registers state R is a total mapping from CPU registers to their value. The memory state M is a total mapping $M\colon A \rightarrow [0 \dots 255]$ of memory addresses to 1-byte memory values, where $A = [0 \dots 2^N - 1]$ is the set of memory addresses, and N is the number of bits used by the CPU for memory addressing. The program counter $pc \in A \cup \{\texttt{halt}\}$ can refer any memory address; `halt` is a special address used to denote the termination of the execution. We assume no distinction between code and data; thus any memory location can potentially be executed. Finally, the exception state $E \in \{\perp$, *illegal instruction*, *division by zero*, *general protection fault*, $\dots\}$ denotes the exception that occurred during the execution of the last instruction; the special exception state $\perp$ indicates that no exception occurred.

The abstract machine that models the CPU is a transition system $(\mathcal{S}, \delta)$. The state-transition function $\delta\colon \mathcal{S} \rightarrow \mathcal{S}$ maps a CPU state $s = (pc, R, M, E)$ into a new state $s' = (pc', R', M', E')$ by executing the current instruction at *pc*. The transition function δ is defined as follows:

$$\delta(pc, R, M, E) \overset{\text{def}}{=} \begin{cases} (pc, R, M, E) & \text{if } pc = \texttt{halt} \vee E \neq \perp, \\ (pc, R, M, E') & \text{if an exception occurs,} \\ (pc', R', M', \perp) & \text{otherwise.} \end{cases}$$

If the program counter points to a valid instruction and the execution of that instruction does not raise any exception, then $\delta(pc, R, M, E) = (pc', R', M', \perp)$. The state of the registers R' and of the memory M' are updated according to the semantics of the executed instruction, the program counter pc' points to the next instruction, and $E' = \perp$. On the other side, if an exception occurs, then $\delta(pc, R, M, E) = (pc, R, M, E')$, with $E' \neq \perp$. An exception indicates that the instruction cannot be executed and, consequently, the program counter, the CPU registers, and the memory remain unvaried. When the last instruction is executed, the program counter is set to `halt`, and from that point on the state of the environment is not updated anymore. The same applies after an exception has occurred.

Having formalised how the CPU executes code, we can now define what it means for $\mathcal{C}_E$ to be a *faithful emulator* of $\mathcal{C}_P$. Intuitively, $\mathcal{C}_E$ faithfully emulates $\mathcal{C}_P$ 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, $\mathcal{C}_E$ *faithfully emulates* $\mathcal{C}_P$ iff:

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

2.3 Fuzzing CPU Emulators

Obviously, proving that $\mathcal{C}_E$ faithfully emulates $\mathcal{C}_P$ is infeasible because of the unmanageable number of states that would have to be tested. For this reason, instead of trying to prove that $\mathcal{C}_E$ faithfully emulates $\mathcal{C}_P$, we relax our goal and try to prove the opposite. That is, we search for an execution state $\bar{s} \in \mathcal{S}$ that demonstrates that $\mathcal{C}_E$ does not faithfully emulates $\mathcal{C}_P$. More formally, $\mathcal{C}_E$ *unfaithfully emulates* $\mathcal{C}_P$ iff:

$$\exists\, \bar{s} \in \mathcal{S} : \delta_{\mathcal{C}_P}(\bar{s}) \neq \delta_{\mathcal{C}_E}(\bar{s}).$$

Because we assume $\mathcal{C}_P$ to be correct, the existence of such a state testifies the existence of a defect in $\mathcal{C}_E$.

Our approach to detect if $\mathcal{C}_E$ is not a faithful emulator of $\mathcal{C}_P$ is based on fuzzing. We generate a synthetic state (or test-case) $s = (pc,\ R,\ M,\ \perp)$ and we set the state of both $\mathcal{C}_P$ and $\mathcal{C}_E$ to s. Then we execute the instruction pointed by *pc* in both $\mathcal{C}_P$ and $\mathcal{C}_E$. At the end of the execution of the instruction, we compare the resulting state. If no difference is found, then $\delta_{\mathcal{C}_P}(s) = \delta_{\mathcal{C}_E}(s)$ holds. On the other hand, a difference in the final states proves that $\delta_{\mathcal{C}_P}(s) \neq \delta_{\mathcal{C}_E}(s)$ and therefore that $\mathcal{C}_E$ does not faithfully emulate $\mathcal{C}_P$.

Figure 1 shows an example of our testing methodology in action. We run two different test-cases, namely s and $\bar{s}$. To ease the representation, in the figure we report only the meaningful state information (three registers and the content of few memory locations) and we represent the program counter by underling the instruction it is pointing to. Furthermore, when the states of the two environments do not differ, we graphically overlap them. The first test-case s (Figure 1(a)) consists in executing the instruction `mov $0x1, %eax`. We execute concurrently this test-case on $\mathcal{C}_P$ and $\mathcal{C}_E$: we set the state of the two environments to s and we execute in both the instruction pointed by the program counter. We observe no difference in their final state. Therefore, we conclude that $\delta_{\mathcal{C}_E}(s) = \delta_{\mathcal{C}_P}(s)$ and that, for the state s, $\mathcal{C}_P$ is faithfully emulated by $\mathcal{C}_E$. The second test-case $\bar{s}$ (Figure 1(b)) consists in executing the instruction `push %fs`, that saves the segment register `fs` on the stack. Although the register is 16 bits wide, the IA-32 specification dictates that, when operating in 32-bit mode, the CPU has to reserve 32 bits of the stack for the store. In the example we observe that $\mathcal{C}_P$ leaves the upper 16 bits of the stack untouched, while $\mathcal{C}_E$ overwrites them with zero (the different bytes are highlighted in the figure). The final state of the two environments differs because the content of their memory differs. Consequently, we have that, for $\bar{s}$, $\delta_{\mathcal{C}_P}(\bar{s}) \neq \delta_{\mathcal{C}_E}(\bar{s})$. That proves that $\mathcal{C}_E$ does not faithfully emulate $\mathcal{C}_P$. It is worth noting that this example reflects a real defect we have found in QEMU using our testing methodology.

3. EMUFUZZER

The development of the fuzzing-based approach just described requires two major efforts. First, as the number of states in which the environment has to be tested is prohibitively large, we have to focus our efforts on a small subset of states. Consequently, we have to carefully craft those states to avoid redundancy and to maximise the completeness of the testing. Second, the detection of deviations in

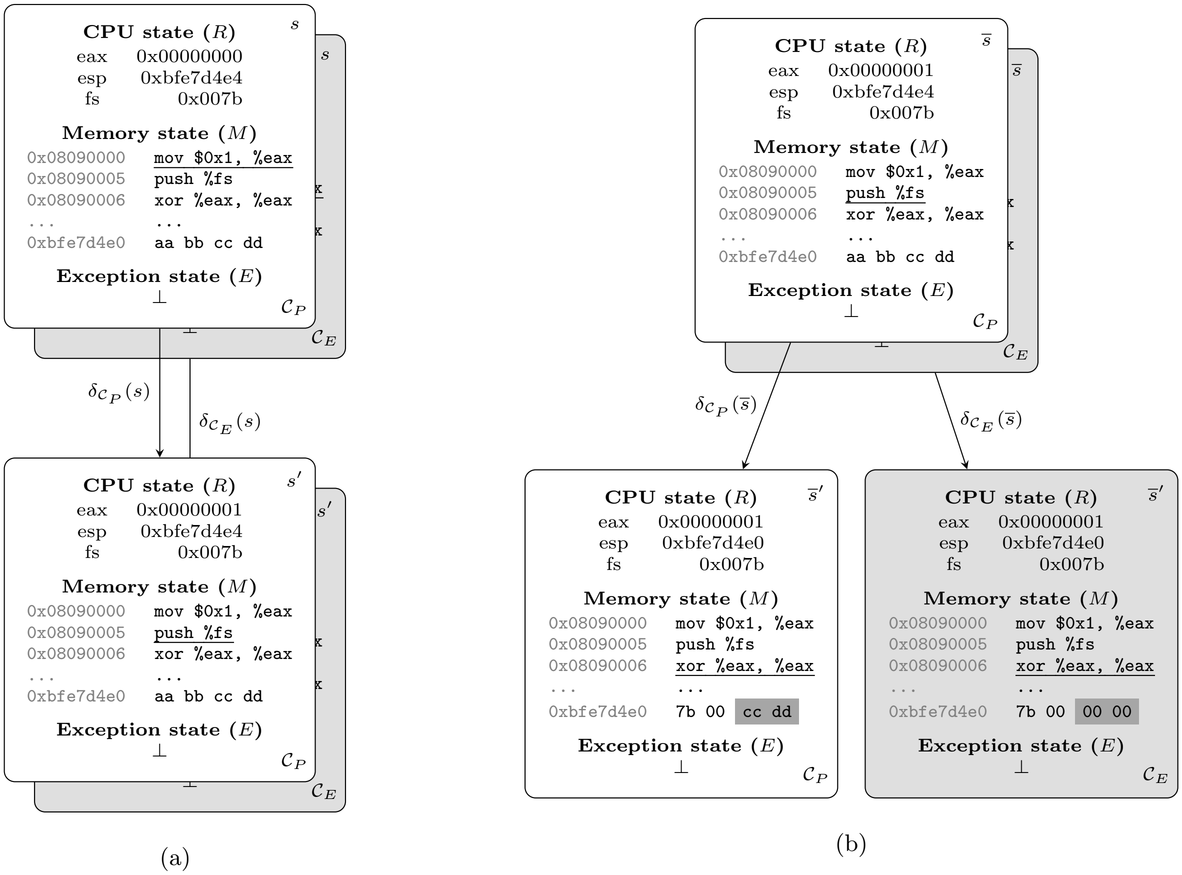


Figure 1: An example of our testing methodology with two different test-cases (s and $\overline{s}$): (a) no deviation in the behaviour is observed, (b) the words at the top of the stack differ (highlighted in gray).

the behaviours of the two environments requires to setup the two and to inspect their state at the end of the execution of each test-case. Thus, we need to develop a mechanism to efficiently initialise and compare the state of the two environments. This section describes the details of our testing methodology.

Although the methodology we are proposing is architecture independent, our implementation, codenamed EmuFuzzer, is currently specific for IA-32. This choice is solely motivated by our limited hardware availability. Nevertheless, minor changes to the implementation would be sufficient to port it to different architectures. To ease the development, the current version of the prototype runs entirely in user-space and thus can only verify the correctness of the emulation of unprivileged instructions and whether privileged instructions are correctly prohibited. EmuFuzzer deals with two different types of emulators: process emulators that emulate a single process at a time (e.g., Valgrind, PIN, and QEMU), and whole-system emulators that emulate an entire system (e.g., BOCHS, Simics, and QEMU[2]).

3.1 Test-case Generation

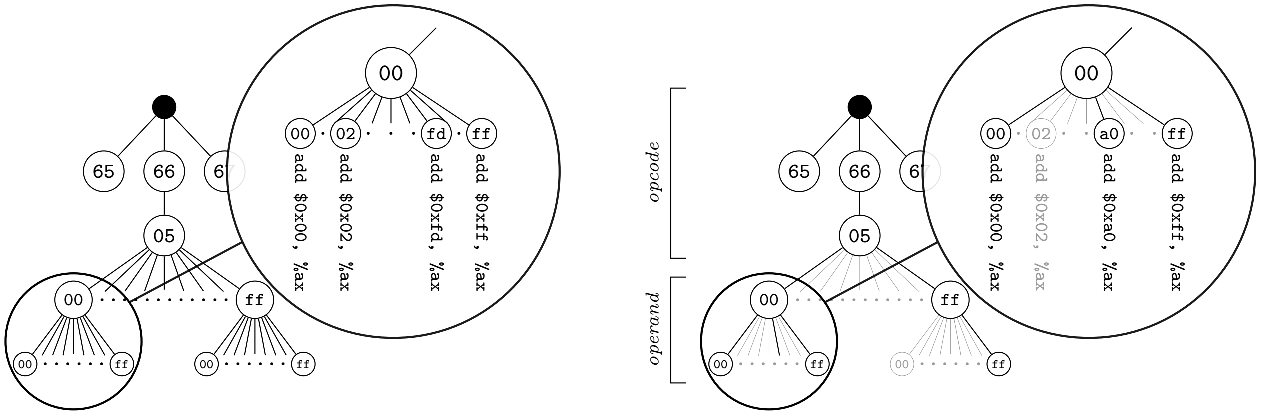
In our testing methodology, the test-cases are merely the states of the environment under testing. For simplicity we consider a test-case as composed by data and code. If $s = (pc, R, M, \perp)$ is the test case, the code consists in the bytes loaded in memory, representing the instruction (or the sequence of instructions) pointed by pc and that will be executed by the CPU. The data of the test-case are R and the remaining bytes of memory. To generate test-cases we adopt two strategies: (I) *random test-case generation*, where both data and code are random, and (II) *CPU-assisted test-case generation*, where data are random, while code is generated algorithmically, with the support of the physical and of the emulated CPUs. The advantage of using two different strategies is a better coverage of the test-case space.

Practically speaking, a test-case consists in a small assembly program, generated with one of the aforementioned techniques. Figure 2 shows a sample test-case (written in C for clarity). This program initialises the state of the environment, by loading the data of the test-case in memory (lines 5–9) and in the CPU (lines 11–13), and subsequently triggers the execution of the code of the test-case (lines 15–16). The program is compiled with special compiler flags to generate a tiny self-contained executable (i.e., that does not use any shared library).

3.1.1 Random Test-case Generation

In random test-case generation, both data and code of the test-case are generated randomly. The memory is initialised by mapping a file filled with random data. For simplicity, the same file is mapped multiple times at consecutive addresses until the entire user-portion of the address space is

[2] QEMU supports both whole-system and process emulation.



(a)

(b)

Figure 3: Example of CPU-assisted test-case generation for the opcode 66 05 **(**mov imm16,%ax**): (a) naïve and (b) optimised generation (paths in gray are not explored).**

```
void main() {
  void *p;
  char code[] = "code of the test-case";

  // Initialise the memory with random data
  for (p = 0x0; p < FILE_SIZE; p += FILE_SIZE) {
    f = open(FILE_WITH_RANDOM_DATA, O_RDWR);
    mmap(p, PAGE_SIZE, ..., MAP_FIXED, f, 0);
  }

  // Initialise the registers with random data
  asm("mov RANDOM, %eax");
  ...

  // Execute the code of the test-case (pc = code)
  ((void(*)()) code)();
}
```

Figure 2: Sample test-case (in C for clarity).

allocated. To avoid a useless waste of memory, the file is lazily mapped in memory, such that physical memory pages are allocated only if they are accessed. The CPU registers are also initialised with random values. As we work in user-space, we cannot allocate the entire address space because a part of it is reserved to the kernel. Therefore, to minimise page faults when registers are used to dereference memory locations, we make sure the value of general purpose registers fall around the middle of the allocated address space. Obviously, code generated with this approach might contain more than one instruction.

3.1.2 CPU-assisted Test-case Generation

A thorough testing of an emulator requires to verify that each possible instruction is emulated faithfully. Unfortunately, the pure random test-case generation approach presented earlier is very unlikely to cover the entire instruction set of the architecture (the majority of CPU instructions require operands encoded using specific encoding and others have opcodes of multiple bytes). Ideally, we would have to enumerate and test all possible instances of instructions (i.e., combinations of opcodes and operands). Clearly this is not feasible. To narrow the problem space, we identify all supported instructions and then we test the emulator using only few peculiar instances of each instruction. That is, for each opcode we generate test-cases by combining the opcodes with some predefined operand values. As in random-test case generation, the data of the test-case are random.

Naïve Exploration of the Instruction Set. Our algorithm for generating the code of a test-case leverages both the physical and the emulated CPUs, in order to identify byte sequences representing valid instructions. We call our algorithm *CPU-assisted test-case generation.* The algorithm enumerates the sequences of bytes and discards all the sequences that do not represent valid code. The CPU is the oracle that tells us if a sequence of bytes encodes a valid instruction or not: sequences that raise illegal instruction exceptions do not represent valid code. We run our algorithm on the physical and on the emulated CPUs and then we take the union of the two sets of valid instructions found. The sequences of bytes that cannot be executed on any of the CPUs are discarded because they do not represent interesting test-cases: we know in advance that the CPUs will behave equivalently (i.e., $E' =$ *illegal instruction*). On the other hand, a sequence of bytes that can be executed on at least one of the two CPUs is considered interesting because it can lead to one of the following situations: (I) it represents a valid instruction for one CPU and an invalid instruction for the other; (II) it encodes a valid instruction for both CPUs but, once executed, causes the CPUs to transition to two different states.

There are other possible approaches to generate the code of test-cases. For example, one can generate assembly instructions and then compile them with an assembler or use a disassembler to detect which sequences of bytes encode a legal instruction. However, limitations of the assembler or of the disassembler negatively impact on the completeness of the generated test-cases. Besides our approach, none of the ones just mentioned can guarantee no false-negative (i.e., that a sequence of bytes encoding a valid instruction is considered invalid).

Optimised Exploration of the Instruction Set. We can imagine to represent all valid CPU instructions as a tree, where the root is the empty sequence of bytes and the nodes on the path from the root to the leaves represent the various bytes that compose the instruction. Figure 3(a) shows an example of such a tree. Our algorithm exploits a particular property of this tree in order to optimise the traversal and to avoid the generation of redundant test-cases. The majority of instructions have one or more operands and thus multiple sequences of bytes encode the same instruction, but with different operands. All such sequences share the same prefix.

As an example, let us consider the 2^{16} sequences of bytes from `66 05 00 00` to `66 05 FF FF` that represent the same instruction, `add imm16,%ax`, with just different values of the 16-bit immediate operand. Figure 3(a) shows the tree representation of the bytes that encode this instruction. The sub-tree rooted at node `05` encodes all the valid operands of the instruction. Without any insight on the format of the instruction, one has to traverse in depth-first ordering the entire sub-tree and to assume that each path represents a different instruction. Then, for each traversed path, a test-case must be generated. Our algorithm, by traversing only few paths of the sub-tree rooted at node `05`, is able to infer the format of the instruction: (I) the existence of the operand, (II) which bytes of the instruction encode the opcode and which ones encode the operand, and (III) the type of the operand. Once the instruction has been decoded (in the case of the example the opcode is `66 05` and it is followed by a 16-bit immediate), without having to traverse the remaining paths, our algorithm generates a minimal set of test-cases with a very high coverage of all the possible behaviours of the instruction. These test-cases are generated by fixing the bytes of the opcode and varying the bytes of the operand. The intent is to select operand values that more likely generate the larger class of behaviours (e.g., to cause an overflow or to cause an operation with carry). For example, for the opcode `66 05`, our algorithm decodes the instruction by exploring only 0.5% of the total number of paths and generates only 56 test-cases. The optimised tree traversal is shown in Figure 3(b), where paths in gray are those that do not need to be explored. The heuristics on which our rudimentary, but faithful, instructions decoder is built on is briefly described later in the next paragraph. It is worth noting that, unlike traditional disassemblers, we decode instructions without any prior knowledge of their format. Thus, we can infer which bytes of an instruction represent the opcode, but we do not know which high-level instruction (e.g., `add`) is associated with the opcode.

CPU-assisted Instruction Decoding. The optimised traversal algorithm just described requires the ability to decode an instruction, and to identify its opcode and operands. Again, for maximum precision, we use the CPU like an oracle: given a sequence of bytes, the CPU tells us if that sequence encodes a valid instruction or not. The decoding is trial-based: we mutate an executable sequence of bytes, we query the oracle to see which mutations are valid and which are not, and from the result of the queries we infer the format of the instruction. Mutations are generated following specific schemes that reflect the ones used by the CPU to encode operands [11]. The idea is that, if a sequence of bytes contains an operand, we expect all the mutations applied to the bytes of the operand and conforming with its encoding scheme to be valid (i.e., the CPU executes only valid mutations). If all the mutations conforming with a particular encoding scheme lead to valid instructions and some mutations generated with the other schemes do not, we conclude that the mutated bytes of the instruction encode the operand and the mutation scheme successfully applied represents the type of the operand. Moreover, the bytes that precede the operand constitute the opcode of the instruction.

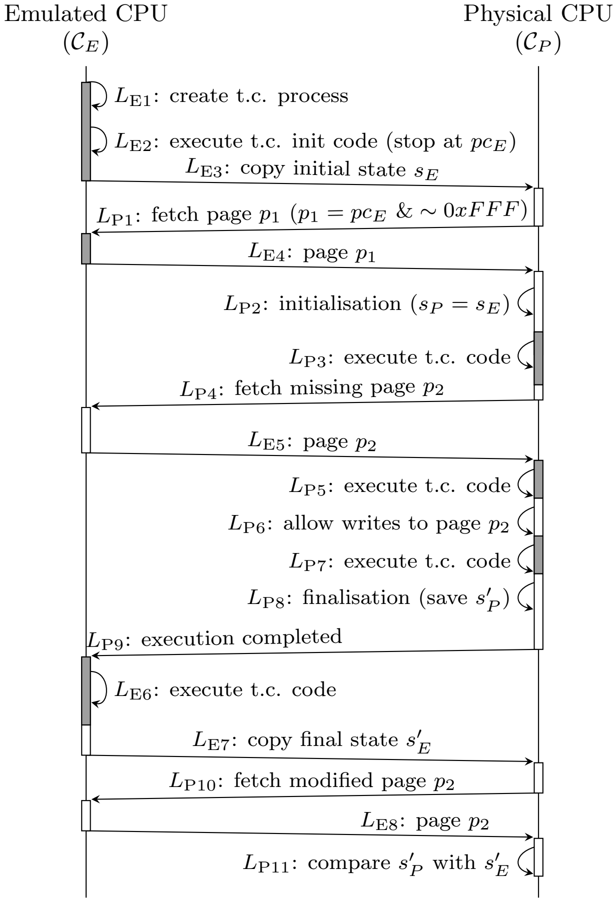

Emulated CPU
($\mathcal{C}_E$)
Physical CPU
($\mathcal{C}_P$)
L_{E1}: create t.c. process
L_{E2}: execute t.c. init code (stop at pc_E)
L_{E3}: copy initial state s_E
L_{P1}: fetch page p_1 ($p_1 = pc_E$ & $\sim 0xFFF$)
L_{E4}: page p_1
L_{P2}: initialisation ($s_P = s_E$)
L_{P3}: execute t.c. code
L_{P4}: fetch missing page p_2
L_{E5}: page p_2
L_{P5}: execute t.c. code
L_{P6}: allow writes to page p_2
L_{P7}: execute t.c. code
L_{P8}: finalisation (save s'_P)
L_{P9}: execution completed
L_{E6}: execute t.c. code
L_{E7}: copy final state s'_E
L_{P10}: fetch modified page p_2
L_{E8}: page p_2
L_{P11}: compare s'_P with s'_E

Figure 4: Logic of the execution of a test-case (t.c., for short). ■ denotes the execution of the test-case and □ denotes the execution of the code of the logic.

3.2 Test-case Execution

Given a test-case, we have to execute it on both the physical and the emulated CPUs and then compare their state at the end of the execution. Our test-cases are small programs that initialise the state of the environment, and transfer the control to the selected sequence of instructions. For this reason, we start by executing the test-case program only in one environment and, as soon as the initialisation of the state is completed, we replicate the status of the registers and the content of the memory pages to the other environment. Then, we execute the code of the test-case in the two environments and, at the end of the execution, we compare the final state. In the current implementation we initially execute the program in the CPU emulator and subsequently replicate the state to the physical CPU. Nevertheless, the core of analysis is performed by the component running in the physical environment. In the remaining of this section we describe the logic of the execution of a test-case and we give details about how we have extended the tested emulators to embed such logic, and how we have developed the module to run a test-case on the physical CPU. For simplicity, the details that follow are specific for the testing of process emulators. Nonetheless, the implementation for testing whole-system emulators only requires the addition of introspection capabilities to isolate the execution of the test-case program [8].

3.2.1 Logic of the Execution of a Test-case

The logic of the execution of a test-case is summarised in Figure 4 and described in detail in the following paragraphs. To facilitate the presentation, we refer to the state of $\mathcal{C}_E$ prior and posterior to the execution of a test-case respectively as $s_E = (pc_E, R_E, M_E, E_E)$ and $s'_E = (pc'_E, R'_E, M'_E, E'_E)$. Similarly, for $\mathcal{C}_P$, we use respectively $s_P = (pc_P, R_P, M_P, E_P)$ and $s'_P = (pc'_P, R'_P, M'_P, E'_P)$.

Setup of the Emulated Execution Environment. The CPU emulator is started and it begins to execute the test-case program ($L_{\mathrm{E}1}$). We let the emulator execute the test-case program until the state of the environment is completely initialised ($L_{\mathrm{E}2}$). In other words, the program is executed without interference until the execution reaches pc_E (i.e., the address of the code of the test case).

Setup of the Physical Execution Environment. When the state of the emulated environment has been setup (i.e., when the execution has reached pc_E), the initial state, $s_E = (pc_E, R_E, M_E, E_E)$, can be replicated into the physical environment. The emulator notifies the module running on the physical CPU and transfers the state of the CPU registers to the latter ($L_{\mathrm{E}3}$). Initially, the exception state E_E is always assumed to be $\perp$. Note that the memory state of the physical CPU M_P is not synchronised with the emulated CPU. At the beginning, only the memory page containing the code of the test-case is copied into the physical environment ($L_{\mathrm{P}1}$ and $L_{\mathrm{E}4}$). The remaining memory pages are instead synchronised on-demand the first time they are accessed, as it will be explained in detail in the next paragraph. At this point we have that $R_E = R_P$, $E_E = E_P = \perp$, but $M_E \neq M_P$ (the only page that is synchronised is the one with the code).

Test-case Execution on the Physical CPU. The execution of the code of the test-case on the physical CPU starts, beginning from program address $pc_P = pc_E$ ($L_{\mathrm{P}3}$). The execution of the code continues until one of the following situations occurs: (I) the execution reaches the last instruction of the test-case; (II) a page-fault exception caused by an access to a missing page occurs; (III) a page-fault exception caused by a write access to a non-writable page occurs; (IV) any other exception occurs. Situation (I) indicates that the entire code of the test-case is executed successfully. That means that all of the instructions of the test-case were valid and did not generate any fatal CPU exception. The first type of page-fault exceptions (II) allows us to synchronise lazily the memory containing the data of the test-case at the first access. During the initialisation phase ($L_{\mathrm{P}2}$) all the memory pages of the physical environment, but that containing the code (and few others containing the code to run the logic), are protected to prevent any access. Consequently, if an instruction of the test-case tries to access the memory, we intercept the access through the page fault exception and we retrieve the entire memory page from the emulated environment ($L_{\mathrm{P}4}$ and $L_{\mathrm{E}5}$). All data pages retrieved are initially marked as read-only to catch future write accesses. After that, the execution of the code of the test-case on the physical CPU is resumed ($L_{\mathrm{P}5}$). The second type of page-fault exceptions (III) allows us to intercept write accesses to the memory. Written pages are the only pages that can differ from an environment to the other. Therefore, after a faulty write operation we flag the memory page as written. Then, the page is marked as writable and the execution is resumed ($L_{\mathrm{P}6}$ and $L_{\mathrm{P}7}$). Obviously, depending on the code of the test-case, situations (II) and (III) may occur repeatedly or may not occur at all during the analysis. Finally, the occurrence of any other exception (IV) indicates that the execution of the test-case cannot be completed because the CPU is unable to execute an instruction. When the execution of the code of the test-case on the physical CPU terminates, because of (I) or (IV), we regain the control of the execution, we immediately save the state of the environment for future comparisons ($L_{\mathrm{P}8}$), and we restore the state of the CPU prior to the execution of the test-case.

Test-case Execution on the Emulated CPU. The execution of the code of the test-case in the emulated environment, previously stopped at pc_E ($L_{\mathrm{E}2}$), can now be safely resumed. The execution of the code in the emulated environment must follow the execution in the physical environment. In the physical environment the state of the memory is synchronised on-demand and thus the initial state of the memory M_E must remain untouched until the physical CPU completes the execution of the test-case. The execution is resumed and it terminates when all the code of the test-case is executed or an exception occurs ($L_{\mathrm{E}6}$).

Comparison of the Final State. Both the emulator and the physical environments have completed the execution of the test-case and thus we can compare their state ($s'_E = (pc'_E, R'_E, M'_E, E'_E)$ and $s'_P = (pc'_P, R'_P, M'_P, E'_P)$). The comparison is performed by the module running in the physical environment. The emulator notifies the other party and then transfers the program counter pc'_E, the current state of the CPU registers R'_E, and the exception state E'_P ($L_{\mathrm{E}7}$). To compare s'_E and s'_P it is not necessary to compare the entire address space: the module running in the physical environment fetches only the content of the pages that have been marked as written ($L_{\mathrm{P}10}$ and $L_{\mathrm{E}8}$). At this point s'_E is compared with s'_P ($L_{\mathrm{P}11}$). If s'_E differs from s'_P, we record the test-case and the difference(s) produced.

3.2.2 Embedding the Logic in the CPU Emulator

The test-case program is run directly in the emulator under analysis. The emulator is extended to include the code that implements the logic of the analysis previously described. We embed the code leveraging the instrumentation API provided by the majority of the emulators. The embedded code serves the following three purposes. First, it allows to intercept the beginning and the end of the execution of each basic block (or instruction, depending on the emulator) of the emulated program. If the code of the test-case contains multiple instructions, all basic blocks (or instructions) are intercepted and contribute to the testing. We assume the code used to initialise the environment is always correctly emulated and thus we do not test it nor we intercept its execution. Second, the embedded code allows to intercept the exceptions that may occur during the execution of the test-case program. Third, it provides an interface to access the values of the registers of the CPU and the content of the memory of the emulator.

3.2.3 *Running the Logic on the Physical CPU*

On the physical CPU, we do not run directly the test-case program, but we run it through a small user-space program that implements the various steps of the analysis described in 3.2.1. An initialisation routine ($L_{\mathrm{P}2}$ in Figure 4), is used to setup the registers of the CPU, to register signal handlers to catch page faults and the other run-time exceptions that can arise during the execution of the test-case, and to transfer the control to the code of the test-case. The code of the test-case is executed as a shellcode [13] and consequently we must be sure it does not contain any dangerous control transfer instruction that would prevent us to regain the control of the execution (e.g., jumps, function calls, system calls). Given the approaches we use to generate the code of the test-cases, we cannot prevent the generation of such dangerous test-cases. Therefore, we rely on a traditional disassembler to analyse the code of the test-case, identify dangerous control transfer instructions, and patch the code to prevent them. At the end of the code of the test-case we append a finalisation routine ($L_{\mathrm{P}8}$ in Figure 4), that is used to save the content of the registers for future comparison, to restore their original content, and to resume the normal execution of the remaining steps of the logic. Exceptions other than page-faults interrupt the execution of the test-case. The handlers of these exceptions record the exception occurred and overwrite the faulty instruction and the following ones with nops, to allow the execution to reach the finalisation routine to save the final state of the environment.

In the approach just described the program implementing the logic and the test-case share the same address space. Therefore, the state of the memory in the physical environment differs slightly from the state of the memory in the emulated environment: some memory pages are used to store the code and the data of the user-space program, through which we run the test-case. If the code of the test-case accesses any of these pages, we would notice a spurious difference in the state of the two environments. Considering that the occurrence of such event is highly improbable, we decided to neglect this problem, to avoid complicating the implementation. To guarantee that at the end of the code of the test-case we are able to regain the control of the execution, we rely on a traditional disassembler to analyse and patch the code of the test-case. If the disassembler failed to detect dangerous control transfer instructions, we could not be able to regain the control of the execution properly. To prevent endless loops caused by failures of this analysis, we put a limit on the maximum CPU time available for the execution of a test-case and we interrupt the execution if the limit is exceeded.

4. EVALUATION

This section presents the results of the testing of four IA-32 emulators with EmuFuzzer: three process emulators (QEMU, Valgrind, and Pin) and a system emulator (BOCHS). We generated a large number of test-cases, evaluated their quality, and fed them to the four emulators. None of the emulators tested turned out to be faithful. In each of them we found different classes of defects: small deviations in the content of the status register after arithmetical and logical operations, improper exception raising, incorrect decoding of instructions, and even crash of the emulator. Our experimental results lead to the following conclusions: (I) developing a CPU emulator is actually very challenging, (II) developers of these software would highly benefit from specialised testing methodology, and (III) EmuFuzzer proved to be a very effective tool for testing CPU emulators.

4.1 Experimental Setup

We performed the evaluation of our testing methodology using an Intel Pentium 4 (3.0 GHz), running Debian GNU/Linux with kernel 2.6.26, as baseline physical CPU. The physical CPU supported the following features: MMX, SSE, SSE2, and SSE3. We tested the latest stable release of each emulator, namely: QEMU 0.9.1, Valgrind 3.3.1, Pin 2.5-23100, and BOCHS 2.3.7. The features of the physical machine were compatible with the features of the tested emulators with few exceptions, which we identified at the end of the testing, using a traditional disassembler, and ignored (for example, BOCHS also supports SSE4).

4.2 Evaluation of Test-case Generation

We generated about 3 million test-cases, 70% of which using our CPU-assisted algorithm and the remaining 30% randomly. We empirically estimated the completeness of the set of instructions covered by the generated test-cases by disassembling the code of the test-cases, by counting the number of different instructions found (operands were ignored), and by comparing this number with the total number of mnemonic instructions recognised by the disassembler. The randomly generated test-cases covered about 75% of the total number of instructions, while the test-cases generated using our CPU-assisted algorithm covered about 62%. Overall, about 81% of the instructions supported by the disassembler were included in the test-cases used for the evaluation. It is worth noting that in several cases our test-cases contained valid instructions not recognised by the disassembler.

The implementation of our CPU-assisted algorithm is not complete and lacks support for all instructions with prefixes. For example, currently our algorithm does not generate test-cases involving instructions operating on 16-bits operands. We have empirically estimated that instructions with prefixes represent more than 25% of the instructions space. Therefore, a complete implementation of the algorithm would allow to achieve a nearly total coverage. We speculate that the high coverage of randomly generated test-cases is due to the fact that the IA-32 instruction set is very dense and consequently a random bytes stream can be interpreted as a series of valid instructions with high probability. Nevertheless, during our empirical evaluation we reached a local optimum from which it was impossible to move away, even after having generated hundreds of thousands of new test-cases. The CPU-assisted algorithm instead does not suffer this kind of problem: a complete implementation would allow to generate a finite number of test-cases exercising all instructions in multiple corner cases.

4.3 Testing of IA-32 Emulators

The four CPU emulators were tested using a small subset ($\sim$10%) of the generated test-cases, selected randomly. The whole testing took about a day, at the speed of around 15 test-cases per second. Table 2 reports the results of our experiments. Behavioural differences found are grouped into three categories: CPU registers state (R), memory state (M), and exception state (E). Differences in the state of the registers are further separated according to the type of the

Deviation type		QEMU		Valgrind		Pin		BOCHS	
		opcodes	*test-cases*	*opcodes*	*test-cases*	*opcodes*	*test-cases*	*opcodes*	*test-cases*
R	*CPU flags*	39	1362	13	684	22	2180	2	2686
	CPU general	3	142	8	141	3	18	8	8
	FPU	179	41738	157	39473	0	0	71	1631
M	*memory state*	34	1586	10	420	0	0	1	2
E	*not supported*	2	1120	334	11513	2	12	0	0
	over supported	97	1859	10	716	0	0	5	8
	other	126	6069	41	6184	20	34	45	113
	Total	**405**	**53926**	**529**	**59135**	**43**	**2245**	**130**	**4469**

Table 2: Results of the evaluation: number of distinct mnemonic opcodes and number of test-cases that triggered deviations in the behaviour between the tested emulators and the baseline physical CPU.

registers: status (*CPU flags*), general purpose and segment (*CPU general*), and floating-point (*FPU*). Differences in the exception state are separated in: legal instructions not supported by the emulator (*not supported*), illegal instructions valid for the emulator (*over supported*), and other deviations in the exception state (*other*). As an example, the last class includes instructions that expect aligned operands but execute without any exception even if the constraint is not satisfied. For each emulator and type of deviation, the table reports the number of distinct mnemonic opcodes leading to the identification of that particular type of deviation (*opcodes*) and the number of test-cases proving the deviation (*test-cases*). It is worth pointing out that different combinations of prefixes and opcodes are considered as different mnemonic opcodes. For each distinct opcode that produced a particular type of deviation, we verified and confirmed manually the correctness of at least one of the results found.

The results demonstrate the effectiveness of the proposed testing methodology. For each emulator we found several mnemonic opcodes not faithfully emulated: 405 in QEMU, 529 in Valgrind, 43 in Pin, and 130 in BOCHS. It is worth noting that some of the deviations found might be caused by too lax specifications of the physical CPU. For example, the manufacturer documentation of the `add` instruction precisely states the effect of the instruction on the status register, while the documentation of `and` states the effect of the instructions only on some bits of the status register, while leaving undefined the value the remaining bits [11]. Our reference of the specification is the CPU itself and consequently, with respect to our definition of faithful emulation, any deviation has to be considered a tangible defect. Indeed, *for each deviation discovered by EmuFuzzer it is possible to write a program that executes correctly in the physical CPU, but crashes in the emulated CPU (or vice versa).* We manually transformed some of the problematic test-cases into such kind of programs and verified the correctness of our claim. The remarkable number of defects found also witnesses the difficulty of developing a fully featured and specification-compliant CPU emulator and motivates our conviction about the need of a proper testing methodology.

The following paragraphs summarise the defects we found in each emulator. The description is very brief because the intent is not criticise the implementation of the tested emulators, but just to show the strength of EmuFuzzer at detecting various classes of defects.

QEMU. A number of arithmetical and logical instructions are not properly executed by the emulator because of an error in the routine responsible for decoding certain encoding of memory operands (e.g., `or %edi, 0x67(%ebx)` encoded as `08 7c e3 67`); the instructions reference the wrong memory locations and thus compute the wrong results. The emulator accepts several illegal combinations of prefixes and opcodes and executes the instruction ignoring the prefixes (e.g., `lock fcos`). Floating-point instructions that require properly aligned memory operands are executed without raising any exception even when the operands are not aligned, because the decoding routine does not perform alignment checking (e.g., `fxsave 0x00012345`). Segments registers, which are 16 bits wide, are emulated as 32-bit registers (the unused bits are set to zero), thus producing deviations when they are stored in other 32-bits registers and in memory (e.g., `push %fs`). Some arithmetic and logical instructions do not faithfully update the status register. Finally, we found sequences of bytes that freeze and others that crash the emulator (e.g., `xgetbv`).

Valgrind. Some instructions have multiple equivalent encodings (i.e., two different opcodes encode the same instruction) but the emulator does not recognise all the encodings and thus the instructions are considered illegal (e.g., `addb $0x47, %ah` with opcode `82`). Several legal privileged instructions, when invoked with insufficient privileges, do not raise the appropriate exceptions (e.g., `mov (%ecx), %cr3` raises an illegal operation exception instead of a general protection fault). On the physical CPU, each instruction is executed atomically and, consequently, when an exception occurs the state of the memory and of the registers correspond to the state preceding the execution of the instruction. On Valgrind instead, instructions are not executed atomically because they are translated into several intermediate instructions. Consequently, if an exception occurs in the middle of the execution of an instruction, the state of the memory and of the registers might differ from the state prior to the execution of the instruction (e.g., `idiv (%ecx)` when the divisor is zero). As in QEMU, some logical instructions do not faithfully update the status register.

Pin. Not all exceptions are properly handled (i.e., trap and illegal instruction exceptions); Pin does not notify the emulated program about these exceptions. Several legal instructions that raise a general protection fault on the physical CPU are executed without generating any exception on

Pin (e.g., `add %ah, %fs:(%ebx)`). When segment registers are stored (and removed) in the stack, the stack pointer is not updated properly: a double-word should be reserved on the stack for these registers, but Pin reserves a single word (e.g., `push %fs`). The FPU appears to be virtualised (i.e., the floating-point code is executed directly on the physical FPU) and, as expected, no deviation is detected in the execution of FPU instructions. As in Valgrind and QEMU, some logical instructions do not faithfully update the status register.

BOCHS. Certain floating-point instructions alter the state of some registers of the FPU and other instructions compute results that differ from those computed by the FPU of the physical CPU (e.g., `fadd %st0, %st7`). If an exception occurs in the middle of the execution of an instruction manipulating the stack, the initial content of the stack pointer corresponds to that we would have if the instruction were successfully executed (e.g., `pop 0xffffffff`). Some instructions do not raise the proper exception (e.g., `int1` raises a general protection fault instead of a trap exception). As in Valgrind, QEMU, and Pin, some logical instruction do not faithfully update the status register, although the number of such instruction is smaller than the number of instructions affected by this problem in the other emulators.

5. DISCUSSION

EmuFuzzer currently works in user-space and thus it can only verify whether unprivileged code is not emulated faithfully, with few exceptions. For example, some unprivileged instructions that access segment registers might not be tested because it is not possible to manipulate properly the value of these registers from user-space. Fortunately, in many cases the values of the segment registers in the emulated and in the physical environments do not need to be manipulated as they already match. Another limitation is that, from user-space, we cannot manipulate control registers and thus we cannot enable supplementary CPU-enforced alignment checking and the other enforcements it offers, which are disabled by default. In the future we plan to port the component running in the physical environment of EmuFuzzer in kernel-space, to be able to perform a more thorough testing. Furthermore, we plan to test new CPU emulators and also to use EmuFuzzer to test the emulation routines adopted by virtual machines to emulate non-virtualisable instructions (i.e., privileged instructions that do not cause a trap if executed with insufficient privileges [28]). As an example, VirtualBox [31] leverages QEMU code for that purpose.

6. RELATED WORK

6.1 Software Testing

Fuzz-testing has been introduced by Miller *et al.* [21], and it is still widely used today for testing different types of applications. Originally, fuzz-testing consisted in feeding applications purely random input data and detecting which inputs were able to crash an application, or to cause unexpected behaviours. Today, this testing methodology is used to test many different types of applications; for example, GUI applications, web applications, scripts, and kernel drivers [5].

As certain applications require inputs with particular format (e.g., a XML document or a well formed Java program), pure randomly generated inputs cannot guarantee a reasonable coverage of the code of the application under analysis. Recently developed testing techniques typically leverage domain specific knowledge and use this knowledge, optionally in tandem with a random component, to drive inputs generation [4, 12, 32]. An alternative approach to improve the completeness of the testing consists in building constraints that describe what properties are required for the input to trigger the execution of particular program paths, and in using a constraint solver to find inputs with these properties [3, 9, 30, 18]. This paper presents a fuzz-testing methodology specific for CPU emulators that leverages both pure random inputs generation and domain knowledge to improve the completeness of the analysis.

The idea of using mechanically generated tests and to compare the behaviour of two components to detect deviations imputable to bugs is known in literature as differential testing [20]. EmuFuzzer adopts differential testing to detect if the tested CPU emulator behaves unfaithfully with respect to the physical CPU emulated.

6.2 Computer Security

CPU emulators are widely used in computer security for various purposes. One of the most common applications is malware analysis [1, 19]. Emulators allow fine-grained monitoring of the execution of a suspicious programs and to infer high-level behaviours. Furthermore they allow to isolate the execution and to easily checkpoint and restore the state of the environment. Malware authors, aware of the techniques used to analyse malware, aim at defeating those techniques such that their software can survive longer. To defeat dynamic behavioural analysis based on emulators, they typically introduce in malware routines able to detect if a program is run in an emulated or in a physical environment. As the average user targeted by the malware does not use emulators, the presence of an emulated environment likely indicates that the program is being analysed. Thus, if the malicious program detects the presence of an emulator, it starts to behave innocuously such that the analysis does not detect any malicious behaviour. Several researchers have analysed state-of-the-art emulators to find unfaithful behaviours that could be used to write specific detection routines [25, 27, 29]. Unfortunately for them, their results were obtained through a manual scrutiny of the source code or rudimentary fuzzers, and thus the results are largely incomplete. The testing technique presented in this paper can be used to find automatically a large class of the unfaithful behaviours that a miscreant could use to detect the presence of an emulated CPU. These information could then be used to harden an emulator, to the point that it satisfies the requirements for undetectability identified by Dinaburg *et al.* [6].

7. CONCLUSIONS

CPU emulators are complex pieces of software. In this paper, we presented a testing methodology for CPU emulators, based on fuzzing. Emulators are tested by generating test-case programs and by executing them on the emulated and on the physical CPUs. As the physical CPU is assumed to follow perfectly the specification, defects in the emulators can be detected by comparing the state of the emulator with that of the physical CPU, after the execution of the test-case program. The proposed methodology has been implemented in a prototype, codenamed EmuFuzzer, and it was used to test four state-of-the-art IA-32 CPU emulators. EmuFuzzer discovered minor and major defects in each of the tested emulators, thus demonstrating the effectiveness of the proposed approach.

8. REFERENCES

[1] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research Annual Conference (EICAR 2006)*, 2006.

[2] 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. USENIX Association.

[3] 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.

[4] 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.

[5] J. DeMott. The Evolving Art of Fuzzing. `http://www.vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf`.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

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

[8] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium, NDSS, San Diego, California, USA*. The Internet Society, Feb. 2003.

[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] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2008. Instruction Set Reference.

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

[13] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

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

[15] H. A. Lichstein. When Should You Emulate? *Datamation*, 1969.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2), 2002.

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

[19] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science. Springer, Sept. 2008.

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

[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. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1978.

[23] NetBSD/amd64. `http://www.netbsd.org/ports/amd64/`.

[24] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, Nov. 2004.

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

[26] D. Quist and V. Smith. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

[27] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proceedings of Information Security Conference (ISC 2007)*. Springer-Verlag, 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 (SSYMM'00)*, Berkeley, CA, USA, 2000. USENIX Association.

[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] Sun Microsystem. VirtualBox. `http://www.virtualbox.org`.

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