Università degli Studi di Milano
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Dottorato di Ricerca in Informatica
XXII ciclo

Threats on Real, Emulated and Virtualized Intel x86 Machine Code Execution

Tesi di
Giampaolo Fresi Roglia

Relatore
Prof. Danilo Bruschi
Coordinatore del Dottorato
Prof. Ernesto Damiani

Anno Accademico 2009/2010

Abstract

The Internet is very popular nowadays, it is used by a continuously growing number of users to transfer confidential data representing a big attractive to cybercriminals. Criminals whose interest is to spread malicious software through different threat vectors aiming to steal those confidential data and sell it in the underground market. Cybercriminals have all the interest in not being detected while perpetrating their intentions. Impeding such threats to spread has become of valuable importance. This goal can be achieved working on the threat vectors cybercriminals use or directly on the threat once identified.

Among threat vectors we can cite application software vulnerabilities which can be abused by malware and malicious users to gain access to systems and confidential data. To be able to impede exploitation of such vulnerabilities, security specialists need to be aware of attack techniques used by malware and malicious users for to be able to design and implement effective protection techniques.

For identifying threats, it is of vital importance to use effective analysis tools which expose no weaknesses to malware authors giving them the chance to evade detection.

This dissertation presents two approaches for testing CPU emulators and system virtual machines which represent a fundamental component of dynamic malware analysis. These testing methodologies can be used to identify behavioural differences between real and emulated hardware. Differences exploitable by malware authors to detect emulation and hide their malicious behaviour.

This dissertation also presents a new exploitation technique against memory error vulnerabilities able to circumvent widely adopted protection strategies like W$\oplus$X and ASLR and the related countermeasure to impede exploitation.

to Paolo and Anna Maria

Contents

1 Introduction

Operating systems and application software become larger and more complex with every release. New features require the addition of new code, which typically contains new bugs. Bugs can be exploited by malicious users and malicious software to gain unauthorised access to systems and sensitive information. New vulnerabilities are discovered on a daily basis on a wide variety of application software. Malicious activities perpetrated through the Internet are becoming quickly a huge security problem, ranging between large-scale social engineering attacks and exploiting critical vulnerabilities. Attack techniques are becoming more sophisticated, malicious software authors are developing an arsenal of offensive techniques ranging from the use of polymorphism, metamorphism and cryptographically strong algorithms to render the task of identifying the threats they pose much more difficult. The goal of malicious activities is turning from disruption of services to the achievement of financial gain.

Figure 1.1 reports malicious code threats detected in 2009 (2,895,802) by Symantec, accounting for over 50% of the total amount of malicious code threats detected over time (approximately 5,700,000). This ever growing number of malicious code threats largely justifies the effort done by researchers for trying to defeat or at least mitigate their diffusion. The methods used by malicious software to spread are known with the name *Threat vectors*. One of the most used and effective threat vector is represented by remotely exploitable vulnerabilities. Malicious code employing such threat vector, according to Symantec account for 24% of the total. Even if remotely exploitable vulnerabilities is not the most used threat vector, it remains one of the most effective for the malware to spread rapidly and infect the maximum number of hosts possible. A notable example of malware using such threat vector is represented by the infamous *Conficker* worm, which, according to PandaLabs was able to infect almost 6% over a sample of 2 million computer analysed in 2008.

The vast majority of reported threats target the so called *Wintel* platform. The reason behind the choice by malware authors resides on the large-scale adoption of such “architecture” by the world-wide market. The Wintel platform represents a software *monocolture*. The large-scale adoption of this platform offers a large attack surface for malicious software to spread, i.e. exploiting a single vulnerability can result in compromising and taking control of a huge number of

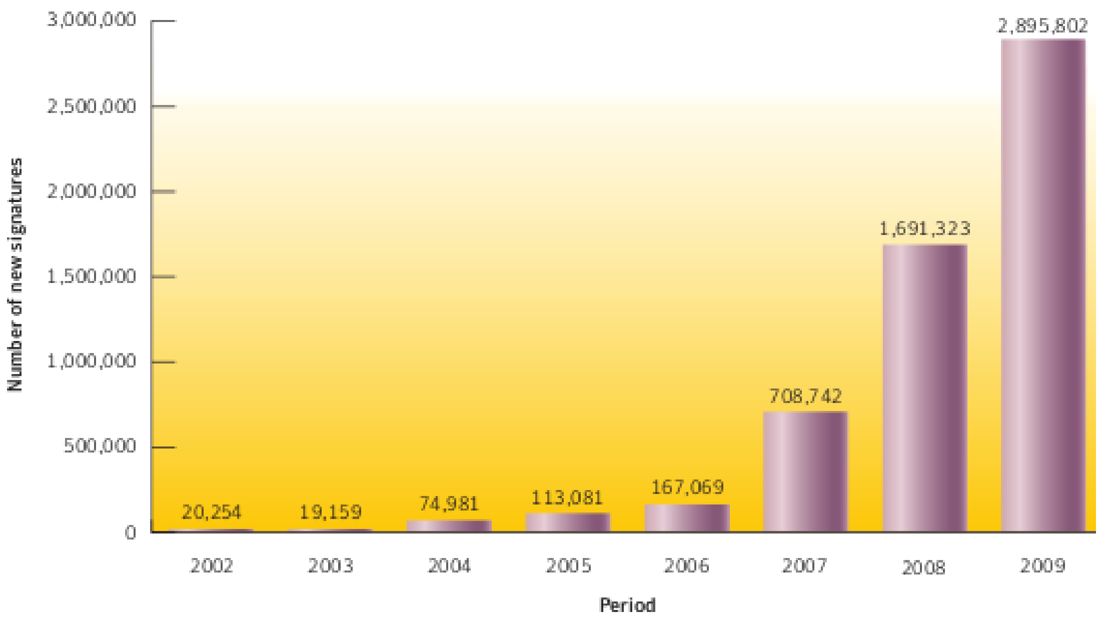


Figure 1.1: *New malicious code signatures*

systems.

The fight against such threats can be done preventing vulnerabilities to be exploited or identifying threats after they already started to spread in the Internet. Prevention can be achieved by users, applying security patches once vulnerabilities are discovered or using anti-virus software and by security specialists, implementing protection techniques able to impede exploitation of vulnerable software.

From the malicious software analyst's point of view, it is of vital importance to recognise as soon as possible such threats as soon as they start to spread for to be able to develop quickly defence strategies against them. For this purpose techniques like static and dynamic binary analysis are employed. With these techniques, the malware analysts aim to recognise malicious behaviours in suspicious programs for providing signatures to intrusion prevention systems and anti-viruses. Unfortunately, malicious software authors are aware of such techniques used against their programs and they develop strategies to evade detection. Static analysis can be easily fooled with techniques like polymorphism and more generally by obfuscating the produced programs. Dynamic analysis by the way has its own limitations: it consists basically of executing the suspicious binary itself and observe its behaviour. For dynamic analysis to guarantee effectiveness, it must guarantee complete code coverage. This implies the observation of every possible execution path of the binary being analysed, objective which can be reduced to the *halting problem* hence impossible to achieve in its full generality. Even with its limits, dynamic analysis is actually employed for run-time detection of malicious behaviours. Analysis of suspicious software is done in isolated environments (sandboxes) to prevent the potential malware to damage the analysis environment. For this purpose malware analysts employ software like system emulators for safely analyse such threats.

Lack of transparency of an emulator w.r.t. the emulated hardware can lead to evasion techniques against dynamic behavioural analysis of malware. If the malicious software were able to detect the presence of an emulator can react by not exposing it's malicious behaviour and stay potentially undetected and free to cause harm for a long time. The in-depth study of Intel x86 ISA (Instruction Set Architecture) gave us the chance of identifying some possible pitfalls which could render the task of correctly implementing an emulator very challenging. The machine-level x86 instruction space is redundant w.r.t. the assembly representation, i.e. an assembly representation can lead to differently encoded but semantically equivalent instructions. Redundancy can be a problem if unlikely used encodings point to the execution of buggy code in the emulator. Typical application software hardly covers the entire machine-level instruction space, so bugs can stay undetected for a long time in absence of the application of specific testing methodologies. Moreover, Intel manuals are written with the application software developer's needs in mind. They do not provide all the details needed for a completely correct implementation of an emulator, hence some side effects exposed by the execution of lots of instructions are left undocumented. The manuals state those effects are to be considered "*undefined*". While this could be sufficient for application software to run correctly, implementing emulators based on those "*specifications*" may result in behavioural differences from the real hardware they are supposed to emulate.

This dissertation provides two specific testing methodologies for spotting differences (not just bugs) between the emulation software and the hardware it wishes to emulate. Information that could be used to harden an emulator to the point it satisfies the requirements for undetectability identified by Dinaburg *et al.* [15].

This dissertation also aims to show state-of-the-art defence techniques against exploitation of memory error vulnerabilities like Address Space Layout Randomisation (ASLR) and Data Execution Prevention (DEP a.k.a W⊕X) can be circumvented under certain circumstances. We provide a new attack technique which even though addresses also issues specific to the *elf* executable format typically used by Unix-like operating systems, is of a more generic applicability, due to the fact that exploits specific characteristics belonging to the underlying hardware architecture, more specifically the Intel x86 ISA, i.e. the ability by the CPU of jumping in the middle of an instruction placed by the compiler, leading to the execution of unintended instructions. In our experience on x86 code, intended instructions (placed intentionally by the compiler) aren't enough for mounting our proposed attack technique.

The first contribution consists of a fully automated testing methodology for CPU emulators, based on fuzzing [41]. 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.

The work led to the implementation of this testing methodology in a prototype for IA-32, code-named EmuFuzzer, and the testing of four state-of-the-art emulators: BOCHS[30], QEMU[4], Pin[34], and Valgrind[44]. 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. The work done with testing these emulators led to the discovery of several deviations in the behaviours of each of them, some of which represent serious defects that might prevent the proper execution of the emulated programs. Example are 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 this testing methodology.

The second contribution of the dissertation consists of a testing methodology specific for system virtual machines. This represents an extension of the work done with EmuFuzzer which leads to the proposal of a much more powerful methodology independent from the technique used by virtual machines to execute guest code. More precisely, the testing technique used with EmuFuzzer is specific for testing CPU emulators (i.e., virtual machines based on CPU emulation) and limited to the testing of emulated user-mode code. The proposed technique can be used to test both user- and system-mode code and thus can also be applied to CPU virtualisers (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, this 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 [59] and differential analysis [39]. Fuzzing is used to generate automatically input states for the testing, and differential analysis to detect anomalous behaviours. Since the expected behaviour corresponds to what can be observed in the CPU of the physical machine, any difference between the behaviours of the physical and virtual machines is clearly anomalous, and consequently a symptom of a defect in the latter.

The second work led to the implementation of a prototype, code-named KEmuFuzzer, for testing virtual machines for the Intel x86 architecture. KEmuFuzzer has been used to test four popular system virtual machines: BOCHS [30], QEMU [4], VMWare [63], and VirtualBox [58]. The first two are based on emulation, while the others are based on direct native execution. In all the virtual machines has been found defects that led to the corruption of the state of the guest operating system or of its applications. The experimental evaluation testified the effectiveness of the proposed methodology and justified the significant effort made to extend it to system-mode code and CPU virtualisers.

The third contribution consists of a new approach to exploit stack-based buffer overflows in programs protected with both W$\oplus$X and ASLR and the related defence technique. The attack is an information leakage attack that exploits information about the random base address at which a library is loaded, available directly in the address space of the process, and is not avoidable. Contrarily to currently existing attack techniques consisting in mounting a brute-force attack, with the proposed technique an attacker can subvert the execution of a vulnerable program and perform a return-to-lib(c) with *surgical precision*, i.e., in a single shot. Furthermore, the technique works independently of the strength of randomisation (i.e., it works on 32 and on 64-bit architecture), and it is applicable to any position dependent executable. The impact of the proposed attack technique is not negligible, since the majority of executables found in modern UNIX distributions belong to this class.

The proposed protection technique is as effective as PIE (the only known protection scheme able to stop the proposed attack at the time of writing) at stopping our attack. Unlike PIE does not require recompilation of any executable, and introduces only negligible overhead. The proposed mitigation technique can be used to protect users of operating systems with ASLR and PIE, but that still have to adopt PIE on large scale (e.g., all GNU/Linux distributions). Moreover, our protection can be used by users of operating systems with ASLR, but lacking PIE (e.g., OpenBSD), and by users of programs with no possibility of recompilation.

1.1 Summary of the contributions

To summarise the dissertation makes the following contributions:

(ISSTA 2009) A testing methodology specific for CPU emulators, based on fuzzing consisting in forcing the emulator to execute 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 [36].

(ISSTA 2010) A testing methodology specific for system virtual machines based on protocol-specific fuzzing and differential analysis consisting in forcing a virtual machine and the corresponding physical machine to execute specially crafted snippets of user- and system-mode code and comparing their behaviours [37].

(ACSAC 2009) A new attack to bypass W⊕X and ASLR based on the leakage of sensitive information about the memory layout of a process and a new protection strategy against this technique which does not require recompilation introducing only a minimal overhead [51].

1.2 Organisation of the dissertation

The dissertation is organised as follows. Chapter 2 provides a brief introduction on some concepts used throughout the dissertation and compares the work presented in the dissertation with the work done by others. Chapter 3 describes a fully automated testing methodology implemented in the prototype code-named EmuFuzzer capable of testing CPU emulators at the user-space level. Chapter 4 describes the extension of the work done with EmuFuzzer for testing both privileged code and user-level code enabling us to test even System Virtual Machines. Chapter 5 describes an attack technique against widely adopted memory-error protections like *ASLR* and *W⊕X* and proposes also an alternative defence strategy against this kind of threats. The dissertation ends discussing possible improvements to the ideas presented and giving concluding remarks in Chapter 6.

2 Background and Related work

This chapter provides a brief introduction on the concepts used throughout the dissertation, briefly describing some basics on Intel x86 ISA, some challenges posed by Intel's architecture on emulation and a brief introduction to techniques used to exploit stack-based buffer overflows, focusing primarily on Intel x86 architecture. Since giving a complete overview of such concepts is beyond the scope of the dissertation, we redirect for this purpose the interested reader to the related literature.

2.1 Overview of Intel x86 ISA

Today, Intel x86 is the most widely adopted computer architecture. It is a complex CISC architecture, with a huge number of different instructions, of variable length, and a myriad of subtle and complex details. All that makes the development of an instruction decoder for this architecture a very tedious and error prone task.

Figure 2.1 depicts the format of an Intel x86 instruction. An instruction is composed of different fields: it starts with up to 4 prefixes, followed by an opcode, an addressing specifier (i.e., `ModR/M` and `SIB` fields), a displacement and an immediate data field [24]. Opcodes are encoded with one, two, or three bytes, but three extra bits of the `ModR/M` field can be used to denote certain opcodes. In total, the instruction set is composed by more than 700 possible values of the opcode field. The `ModR/M` field is used in many instructions to specify non-

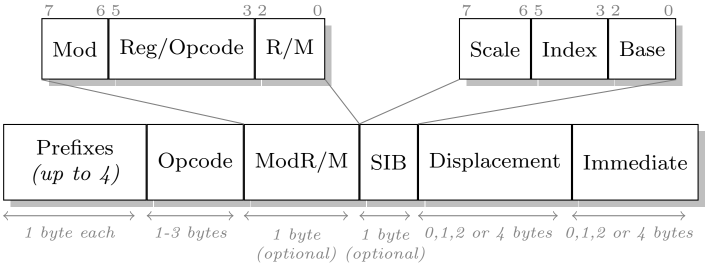


Figure 2.1: *Intel x86 instruction format*

implicit operands: the Mod and R/M sub-fields are used in combination to specify either registry operands or to encode addressing modes, while the Reg/Opcode sub-field can either specify a register number or, as mentioned before, additional bits of opcode information. The SIB byte is used with certain configurations of the ModR/M field, to specify base-plus-index or scale-plus-index addressing forms. The SIB field is in turn partitioned in three sub-fields: Scale, Index, and Base, specifying respectively the scale factor, the index register, and the base register. Finally, the optional addressing displacement and immediate operands are encoded in the Displacement and Immediate fields respectively. Since the encoding of the ModR/M and SIB bytes is not trivial at all, the Intel x86 specification provides tables describing the semantics of the 256 possible values each of these two bytes might assume.

2.1.1 Challenges on x86 emulation

In Computer Science, the term "*emulator*" is typically used to denote a software that simulates a hardware system [33]. 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.

Emulating an x86 CPU means facing the complexity of its instruction set which is already challenging by itself. It is easy to see that elementary decoding operations, such as determining the length of an instruction, require to decode the entire instruction format and to interpret the various fields correctly.

The advent of several instruction extensions (e.g., Multiple Math eXtension (MMX) and Streaming SIMD Extensions (SSE)) made the instruction decoding process even more complicated. As an example, consider the byte strings `f3 ae` and `f3 0f e6`, encoding respectively the instructions `rep scasb` and `cvtdq2pd`. The byte `f3` is a prefix that is found in both instructions, but it serves two different purposes: it represents a `rep` prefix (to repeat the execution of an instruction) in the first case and a mandatory prefix for SSE instructions in the second case. Therefore, an instruction decoder has to consider that the prefix `f3` has to be interpreted differently, according to the subsequent sequence of bytes. The decoder must treat the prefix as a `rep` only when it is followed by a sequence of bytes that encodes a string or I/O operation (e.g., `scasb`), and as a preamble for SSE instructions otherwise (e.g., `cvtdq2pd`).

Intel x86 instruction set has some redundancy w.r.t. the assembly representation of each instruction. Redundancy exists at the opcode level: consider for example the assembly code `test byte [eax], 0x0`. This instruction can be specified with two different opcodes `f6/0` and `f6/1`, resulting in the encodings: `f6 00 00` and `f6 08 00`. It is worth noting the latter representation is undocumented in Intel's manuals while Amd manuals report it.

We can find redundancy also in the memory addressing part: consider for example the assembly code `mov [ecx],eax`. Assemblers typically encode such instruction with the byte sequence `89 01`, where `89` is the opcode of the `mov` instruction and `01` encodes both the memory reference `[ecx]` and the register `eax`. Playing with the `ModR/M` and `SIB` bytes, we can find alternative representations of the same instruction: (1) `89 04 21`, (2) `89 04 61`, (3) `89 04 a1` and (4) `89 04 e1`.

Concluding, Intel x86 reference documentation sometimes lacks proper specifications for certain instructions, and states that others may have undefined effects in certain corner-cases. As an example, the effects on the `OF` flag belonging to the status register `EFLAGS` after the execution of the instruction `shl dword [eax], cl` (shift to the left the double-word pointed by `eax` by the amount specified in the `cl` register), when `cl` is different than 1, must be considered "*undefined*".

2.2 Software Testing

Fuzz-testing has been proposed by Miller *et al.* in 1990 [41], 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 [13, 25, 59]. Another approach consists of building constraints that describe what properties are required for the input to trigger execution of particular program paths, and then use a constraint solver to find inputs with these properties [10, 20, 53, 35].

Differential testing The idea of detecting software defects by comparing the behaviour of two or more software components for the same input is known as differential testing [39]. Differential testing has previously been used in a variety of contexts, including computer security [7], flash file systems [22], and grammar-driven functionality [29]. In [56] 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.

2.3 Emulators in malware analysis

CPU emulators are widely used in computer security for various purposes. One of the most common applications is malware analysis [3, 38]. Emulators allow fine-grained monitoring of the execution of a suspicious program and to infer high-level behaviours. Furthermore they allow to isolate 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 for their software to survive longer. To defeat dynamic behavioural analysis based on emulators, they typically introduce 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 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 [45, 49, 52]. 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 the dissertation can be used to find automatically a large class of unfaithful behaviours 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 it satisfies the requirements for undetectability identified by Dinaburg *et al.* [15].

2.4 Buffer overflow

In high level languages like C, a buffer is declared as an array of data which can be of different types and different lengths. At the machine level, buffers loose these information and can be viewed roughly as a sequence of bytes with unspecified length. It is on a programmer's responsibility to keep track of each buffer length to avoid filling buffers with data beyond their capacity. When a program lacks this kind of checks, a buffer overflow condition can occur. Buffer overflows can be classified in many types, depending on where the declared buffer resides. We can have stack-based, heap-based and static data buffer overflows if the buffer resides respectively in the stack, heap or static data sections of a binary executable. Writing data beyond the buffer length leads to data corruption. The ability by an attacker of corrupting data such as code pointers can lead to arbitrary code execution if such code pointers are used afterwards.

Stack-based buffer overflow is probably the most common and well-known buffer overflow vulnerability. Exploitation happens when a buffer stored in the stack has been filled beyond it's limits corrupting code pointers or data pointers like the saved return address of the calling function or the saved frame pointer. The first known techniques used for exploiting stack-based buffer overflow vulnerabilities involved the injection of executable code in the vulnerable buffer as long as carefully crafted data to overwrite the saved instruction pointer. The saved instruction pointer was typically filled with the address of the buffer itself, leading to execution of injected code, hence giving full control of the vulnerable program by the attacker. For such a technique to be successful, the attacker needs to know in advance the address of the overflown buffer (or at least an approximation) and the process must have the ability to execute code meant to be data by the application.

W⊕X is a defence strategy meant to prevent execution of injected code [61]. It is implemented marking each memory page meant to contain machine code as executable and read-only while marking each memory page meant to contain mutable data as read-write but not executable. Briefly no memory page is allowed to be both executable and writable at the same time. W⊕X was inspired by the belief: if a process were unable to execute attacker-supplied code, the attacker can no more force a vulnerable process to do arbitrary computations. Such belief was later proven to be false by techniques like return-to-libc and it's generalisation: return-oriented programming.

Return-to-libc is an attack technique developed to exploit vulnerabilities like stack-based buffer overflows and circumvent defence strategies like W⊕X [14]. With W⊕X the attacker can no longer inject it's own code. The usual exploit technique involving injection of the so-called shellcode (code meant to spawn a shell back to the attacker) can no longer be applied. However the attacker can still reliably corrupt code pointers to point to existing code. More precisely the attacker can execute useful functions belonging to the ubiquitous libc library, e.g. *system* to obtain a shell. Return-to-libc was believed to be limited as an attack technique for the inability by an attacker to execute arbitrary computations.

Return oriented programming is a generalisation of the return-to-libc technique. This technique was discovered by Shacham [54] and aims to prove the feasibility of arbitrary computation with no code injection at all. This technique consists of reusing small code snippets already present in the vulnerable process' address-space and ending with a `ret` instruction. Each code sequence found accomplishes a single and simple task but glued together by means of carefully constructing a particular stack layout, they can be used to accomplish more complex tasks such as load/store operations, arithmetic operations or operations affecting the control flow such as direct or indirect jumps. Code sequences glued together to implement one of the aforementioned tasks are called "*gadgets*". The author has been able to identify enough gadgets sufficient to construct a Turing-complete machine, hence the possibility of doing arbitrary computations. Initially this technique was believed to be applicable only on CISC-based CPUs like x86 are, due to the ability to execute unaligned instructions, more precisely the ability of the CPU to jump in the middle of an instruction, leading to the execution of unwanted code. Later, Buchanan *et al.* [9] generalised the same technique to be applied on RISC architectures as well.

Address Space Layout Randomisation (ASLR) is a technique developed with the intention of hindering attacks like buffer overflow exploits [60]. For a successful exploitation of a buffer overflow, an attacker needs to know in advance some details on the layout of the vulnerable process. With the simplest stack-based buffer overflow technique the attacker needs to know the base address of the stack section, while for mounting a return-to-libc attack, he needs to know the base address of the libc library.

ASLR consists of randomising the base address of each section belonging to a process. In the presence of ASLR an attacker is no more able to know the base address of those sections since they are randomly chosen and they change at each execution of the vulnerable process. The only option for an attacker to successfully exploit these kind of vulnerabilities is by means of brute-force. However the most widely adopted incarnation of ASLR avoids randomising the base address of the code section belonging to the executable binaries, thus giving the attacker a chance to reliably jump to existing code.

Researchers have demonstrated that W⊕X and ASLR can be defeated by patient attackers [55]. The state-of-the-art approach to exploit stack-based buffer overflows on systems protected with W⊕X and ASLR involves mounting a return-to-lib(c) attack [14] repeatedly, in a brute-force fashion. Indeed, on 32-bit architectures (e.g., Intel x86) ASLR is weak because of low randomisation entropy. Hence, with a relatively small number of attempts an attacker can guess the base random address at which a certain library is loaded and then successfully mount a return-to-lib(c) attack. However, a brute-force attack can easily rise alarms (e.g., because of a large number of crashes) and automatic mechanisms can be used to impede the attacker [23].

Position Independent Executables (PIE) RedHat extended the idea of position independent code to executables. Like shared objects, position independent executables (PIE) can be loaded at arbitrary memory locations [62]. In this dissertation, we present two variants of a new attack technique to exploit stack-based buffer overflow vulnerabilities. Both variants of the attack explained in chapter 5 exploit the lack of randomisation affecting some `elf` sections, e.g. `.text`, `.plt` and `.got`. None of the variants of our attack can be applied to position independent executables because, as for randomised libraries, the address of code chunks varies from one execution to another. Therefore, guessing the absolute address of a code chunk in an executable becomes as hard as reusing the code of a shared library. To further complicate the exploitation, position independent executables can also be used to construct self-randomising executables [6], executables that rearrange automatically, at each execution, the disposition of their functions.

Code randomisation Bhaktar *et al.* [5] proposed a randomisation scheme that uses binary rewriting to periodically re-obfuscate an executable, including the layout of the code section. The randomisation of the code section could prevent an attacker from using code chunks available in the executable. However, since re-obfuscation is periodic, a local attacker accessing the executable on disk can successfully mount both variants of our attack, within the time window in which the executable does not change.

GOT-related protections Xu *et al.* [65] designed a runtime system that randomises the location of the GOT and patches the PLT accordingly. This system essentially just adds a fake layer of security: the sensitive information we abuse in our proposed attack technique, (the content of the GOT) is stored at a random location, but the address of this location remains accessible in memory (in the PLT). Through our attack it is possible to dereference the PLT, discover the address of the GOT, and then overwrite or dereference any GOT entry. However, to perform GOT overwriting, the code chunks necessary for a dereference must be available in the executable. Recent versions of `binutils` include support for producing executables with a read-only GOT [19]. A similar protection could also be implemented at runtime, by adopting a system like the one proposed by Xu *et al.* Clearly such protection prevents our GOT overwriting attack, but it cannot mitigate the first variant of the attack. Unfortunately, despite the fact that this protection has been available in `binutils` for years, our experimental analysis demonstrated that this protection is not yet adopted by any distribution.

N-Variant system A completely different approach to detect memory corruptions is the N-Variant system [11]. The idea is to run n different instances of the same program with diverse memory layouts, obtained using ASLR. Any successful attack will work only on one of the instances and will cause all the other instances to crash because the attack must be tailored to a particular process layout. This idea has been further extended in [8].

2.4.1 Notes on x86 unwanted code

Our proposed attack technique is based return-oriented programming [54]. However his technique does not take into account ASLR protected binaries and cannot be applied to the code section of a generic binary executable due to the lack of a sufficient number of usable code chunks, necessary to construct a Turing-complete machine. In our attack we look for code chunks in the code section of an elf binary. Due to the limited amount of code we can abuse w.r.t. code available in the libc library, more precisely due to the presence of very few code chunks ending with a `ret` instruction, we found to have a small chance of being successful in mounting our attack by looking for wanted code only (code intentionally placed by the compiler). The chance of success heavily increases if we take into account unwanted code too.

Unwanted code shows up by the ability of x86 CPUs to jump in the middle of an instruction, leading to a completely different computation.

```
8902    mov [edx],eax
83C42C  add esp,byte +0x2c
89F0    mov eax,esi
5B      pop ebx
5E      pop esi
5F      pop edi
5D      pop ebp
C3      ret
```

Figure 2.2: *function epilogue*

```
0283C42C89F0 add al,[ebx-0xf76d33c]
5B           pop ebx
5E           pop esi
5F           pop edi
5D           pop ebp
C3           ret
```

Figure 2.3: *unwanted code*

As an example, consider the code sequence in Figure 2.2. If we start disassembling in the middle of the first instruction, we obtain a different code sequence depicted in Figure 2.3. The former representing a typical function epilogue, the latter representing the unwanted code we abuse.

Despite the ability of Buchanan *et al.* [9] to apply return-oriented programming to RISC architectures like SPARC, which avoids the presence of unwanted code, we argue unwanted code is essential for our attack technique to be successful.

3 EmuFuzzer: a testing methodology for CPU emulators

This chapter 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 (code-named 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.

3.1 Introduction

Defects in emulators may affect the execution of benign and malicious software. Emulators started to be used in the field of malware analysis for facilitating the analysis of suspicious software as they provide an isolated environment to safely execute potentially harmful programs. Dynamic malware analysis aims to spot harmful behaviours in software programs for identifying such threats and be able to react against them. Malicious software writers learnt to use techniques to fool malware analysis and evade detection. One of the evasion techniques they employ consists of detecting the presence of an emulator for not exposing their malicious behaviour to the potential analysis environment. To avoid this problem, emulators must be transparent w.r.t. the hardware they wish to emulate. Emulators are complex software, this is especially true if they have to emulate complex CISC architectures like x86. For their complexity they are also unlikely free of bugs.

Although several good tools and debugging techniques exist [42], 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.

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 3.1: *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).*

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 3.1[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 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 [17, 45, 48, 49, 52]. 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. Apart from spotting semantic differences between instructions executed on the real CPU and by an emulated one, there are other ways to detect the presence of an emulator. Emulating a single instruction, tipically costs several CPU cycles. A malicious program willing to detect emulation could measure the time needed for a particular instruction sequence to execute. Detection could happen if the measured time is over some threshold. Such thresholds can be empirically obtained measuring time needed for direct CPU execution. Clearly, the work presented in the dissertation does not cover such collateral effects exposed by the emulation, being focused only on

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

instruction semantics.

This chapter makes the following contributions:

- An automated testing methodology specific for CPU emulators, based on fuzzing and differential testing. We generate a large number of test-cases and run these test-cases on both the emulated and the physical CPU. We compare the configuration of the two environments looking for differences.
- A prototype implementation for IA-32 of the aforementioned testing technique code-named EmuFuzzer.
- Evaluation results of the prototype applied to four state-of-the-art emulators (BOCHS, QEMU, Pin, and Valgrind), allowed us to find bugs in all of them.

3.2 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).

3.2.1 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 $\mathcal{C}_P$,

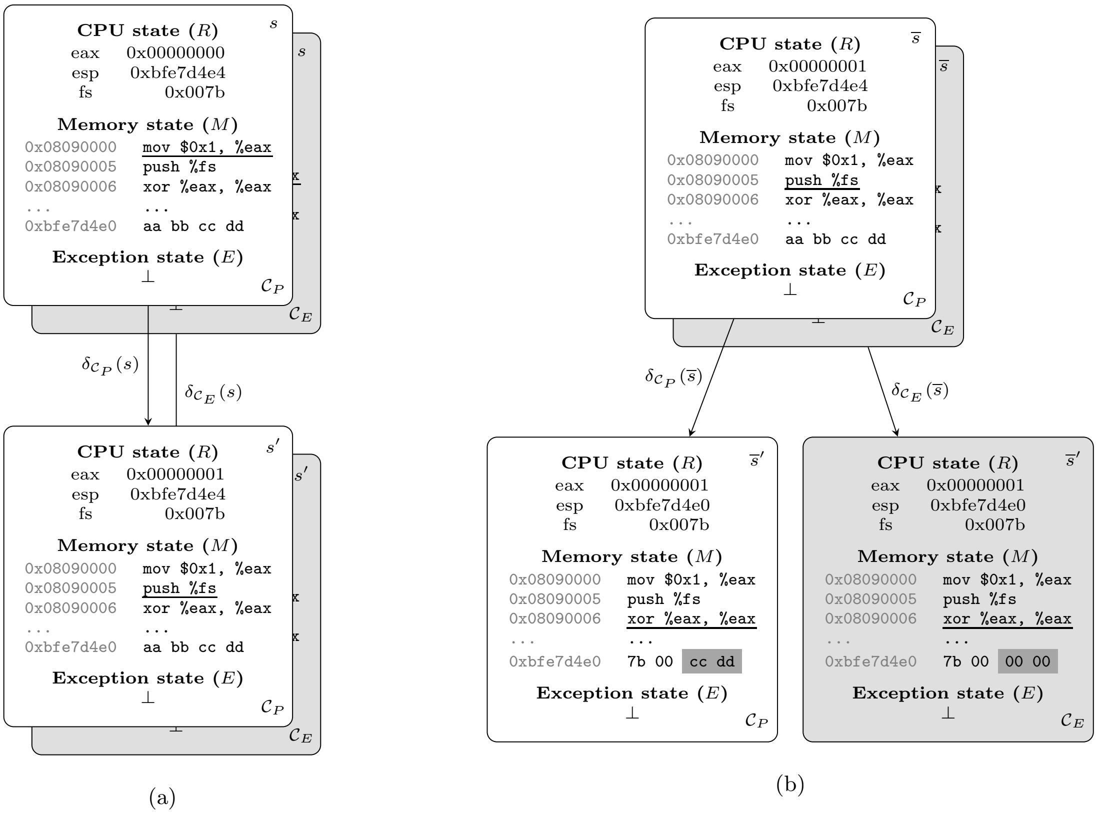


Figure 3.1: *An example of our testing methodology with two different test-cases (s and $\overline{s}$): (a) no deviation in the behaviour is observed, (b) the words at the top of the stack differ (highlighted in gray).*

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 $M\colon A \to [0 \ldots 255]$ of memory addresses to 1-byte memory values, where $A = [0 \ldots 2^N - 1]$ 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 $E \in \{\perp,\ \textit{illegal instruction},\ \textit{division by zero},\ \textit{general protection fault},\ \ldots\}$ 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 $\delta\colon \mathcal{S} \to \mathcal{S}$ 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:

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

3.2.2 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) $s = (pc, R, M, \bot)$ 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 3.1 shows an example of our testing methodology in action. We run two different test-cases, namely s and $\overline{s}$. 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 underlining 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 3.1(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 $\overline{s}$ (Figure 3.1(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 $\overline{s}$, $\delta_{\mathcal{C}_P}(\overline{s}) \neq \delta_{\mathcal{C}_E}(\overline{s})$. 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.

3.3 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 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]).

3.3.1 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 3.2 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).

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

```
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)();
}
```

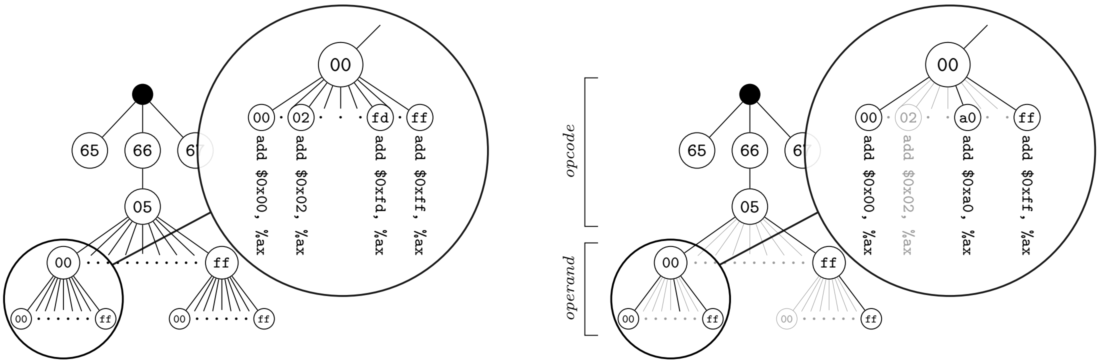
Figure 3.2: *Sample test-case (in C for clarity).*

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 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.

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



(a)

(b)

Figure 3.3: *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).*

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 3.3(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 starting 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 3.3(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 3.3(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. If the instruction is valid, the CPU tells us it's length. Apart from *length-changing prefixes*, given a fixed opcode, the only length-changing part of an instruction is the memory-addressing part. Once we identify the opcode and, if present, the memory addressing part, we can infer the operand type knowing it's length. 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 [24]. 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.

Decoding Instruction Length Given an arbitrary sequence of bytes $B = b_1 \ldots b_n$, the first goal is to detect if the bytes represent a valid instruction. Then, for valid instructions, we have to infer their length. Our decoder exploits the fact that the CPU fetches from the memory the bytes of the instruction and decodes them incrementally. The decoder executes the input string B in a specially crafted execution environment, such that every fetch of the bytes composing the instruction can be observed.

The decoder initially partitions B into subsequences of incremental length ($B_1 = b_1$, $B_2 = b_1 b_2$, $\ldots$, $B_n = b_1 \ldots b_n$) and then executes one subsequence after another, using single-stepping. Since the goal is to intercept the fetch of the various bytes of the instruction, the i^{th} subsequence B_i (with $i = 1 \ldots n$) is placed

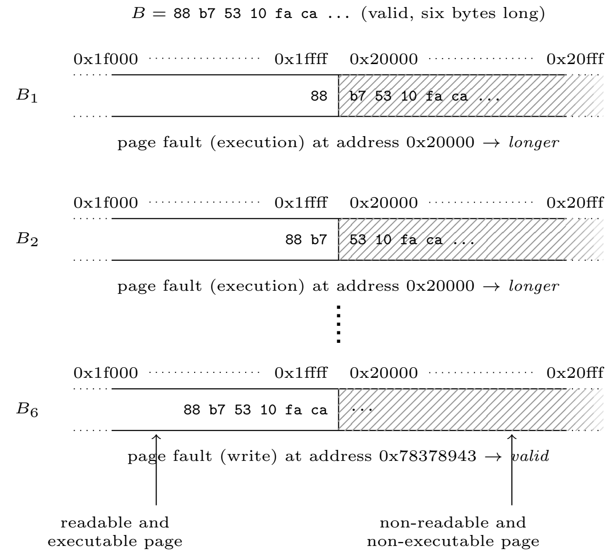

B = 88 b7 53 10 fa ca ... (valid, six bytes long)
0x1f000
0x1ffff
0x20000
0x20fff
B1
88
b7 53 10 fa ca ...
page fault (execution) at address 0x20000 → longer
B2
88 b7
53 10 fa ca ...
page fault (execution) at address 0x20000 → longer
B6
88 b7 53 10 fa ca
...
page fault (write) at address 0x78378943 → valid
readable and executable page
non-readable and non-executable page

Figure 3.4: *Computation of the length of instructions using our CPU-assisted instruction decoder.*

in memory such that it overlaps two adjacent memory pages, m and m'. The first i bytes are located at the end of m, and the remaining bytes at the beginning of m'. The two pages have special permissions: m allows read and execute accesses, while m' prohibits any access. When the instruction is executed, the i bytes in the first page are fetched incrementally by the CPU. If the instruction is *longer* than i bytes, when the $(i+1)^{th}$ byte is fetched the CPU raises a page fault exception (where the faulty address corresponds to the base address of m' and the cause of the fault is an instruction fetch) because the page containing the byte being read, m', is not accessible. If the instruction is i bytes long instead, the CPU executes the instruction without accessing the bytes in m'. In such a situation the instruction can be both valid and invalid. The instruction is *valid*, and i bytes long, if it is executed without causing any exception; it is also valid if the CPU raises a page fault or a general protection fault exception. A page fault exception occurs if the instruction tries to read or write data from the memory (in this case the faulty address does not correspond to the base address of m'); a general protection fault exception is raised if the instruction has improper operands (e.g., it expects aligned operands but alignment is not respected). The instruction is *invalid* instead, if the CPU raises an illegal instruction exception. If the instruction is either valid or invalid the decoder returns, otherwise, it repeats the process with the next subsequence, B_{i+1}.

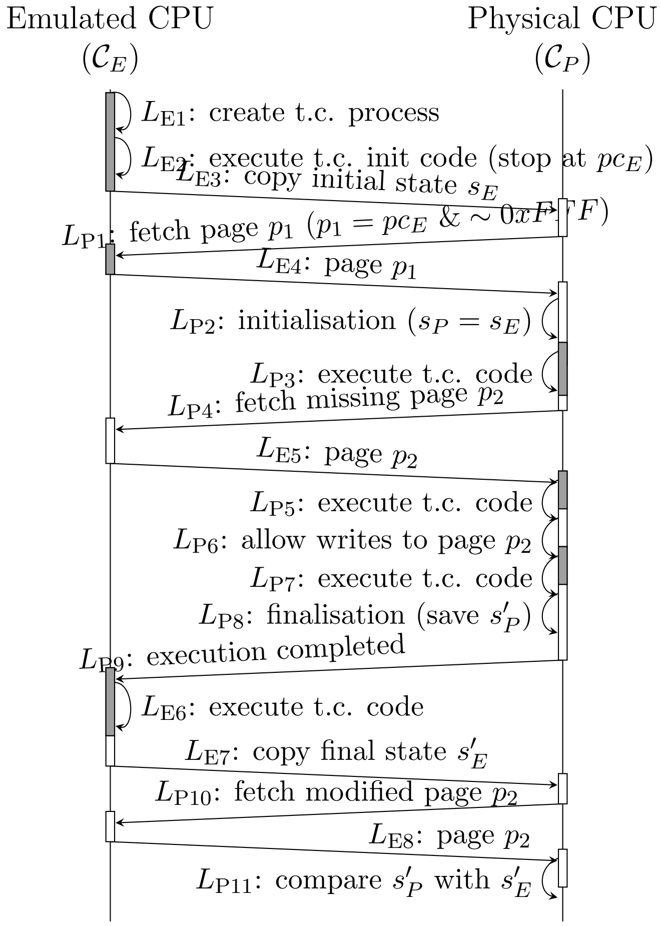

Emulated CPU
($\mathcal{C}_E$)
Physical CPU
($\mathcal{C}_P$)
L_{E1}: create t.c. process
L_{E2}: execute t.c. init code (stop at pc_E)
L_{E3}: copy initial state s_E
L_{P1}: fetch page p_1 ($p_1 = pc_E$ & $\sim 0xFFF$)
L_{E4}: page p_1
L_{P2}: initialisation ($s_P = s_E$)
L_{P3}: execute t.c. code
L_{P4}: 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 3.5: *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.*

3.3.2 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 [18].

Logic of the Execution of a Test-case

The logic of the execution of a test-case is summarised in Figure 3.5 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_{E1}). We let the emulator execute the test-case program until the state of the environment is completely initialised (L_{E2}). 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_{E3}). 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_{P1} and L_{E4}). 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_{P3}). 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_{P2}) 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_{P4} and L_{E5}). 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_{P5}). 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_{P6} and L_{P7}). 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_{P8}), 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_{E2}), 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_{E6}).

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_{E7}). 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_{P10} and L_{E8}). At this point s'_E is compared with s'_P (L_{P11}). If s'_E differs from s'_P, we record the test-case and the difference(s) produced.

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.

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 3.3.2. An initialisation routine (L_{P2} in Figure 3.5), 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 [26] 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_{P8} in Figure 3.5), 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.

3.4 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.

Deviation type		**QEMU**		**Valgrind**		**Pin**		**BOCHS**	
		opcodes	*testc*	*opcodes*	*testc*	*opcodes*	*testc*	*opcodes*	*testc*
R	*CPU flags*	39	1362	13	684	22	2180	2	2686
	CPU gp	3	142	8	141	3	18	8	8
	FPU	179	41738	157	39473	0	0	71	1631
M	*mem state*	34	1586	10	420	0	0	1	2
E	*not supp.*	2	1120	334	11513	2	12	0	0
	over supp.	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 3.2: *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.*

3.4.1 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).

3.4.2 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.

3.4.3 Testing of IA-32 Emulators

The four CPU emulators were tested using a small subset (∼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 3.2 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 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 [24]. 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.

3.5 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. We addressed the limits of EmuFuzzer in the next chapter, in which we propose a different approach able to test also privileged code in different execution modes.

4 KEmuFuzzer: a methodology for testing System Virtual Machines

In this chapter we present a methodology specific for testing system virtual machines. This methodology is an improvement over the one proposed in the previous chapter. Like in EmuFuzzer the testing 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 behaviours. We have developed a prototype, code-named 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 discovering defects in all of them.

4.1 Introduction

In the previous chapter we proposed a testing methodology for CPU emulators. Such methodology can be used to test only user-mode code and can't be used for testing privileged code. System virtual machines, especially those relying on direct binary execution can't take advantage of such testing methodology for they execute user-mode code directly using the real CPU so, for the way the testing works, they must appear correct by definition. However, system virtual machines not leveraging virtualisation extensions like Intel's VT-x technology, include some software emulation in them. Emulation is necessary for to be able to simulate the execution of privileged code while in user-mode. For example, Virtualbox includes part of the code of QEMU for such purpose.

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 optimisation. On the other hand, typical usages of system virtual machines are resources consolidation [1], applications provisioning, simplification of maintenance, system integration [40], development [21], 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. 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 virtualisation and to maximise the amount of guest code that can be run natively [43]. 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 [64], their thorough testing must be taken very seriously.

This chapter makes the following contributions:

- An automated testing methodology specific for system virtual machines based on protocol-specific fuzzing and differential testing. We present a more powerful methodology than the one we used with EmuFuzzer. The new approach enabled us to test user- and system-mode code thus can be applied to CPU virtualizers.

- A prototype implementation for IA-32 of the proposed testing methodology code-named KEmuFuzzer.

- Evaluation results of KEmuFuzzer, applied to four popular system virtual machines (BOCHS, QEMU, VMWare and VirtualBox), allowed us to find defects in all of them.

4.2 Overview

We describe the approaches used in system virtual machines to simulate the physical CPU, we introduce a property of virtual machines we called *transparency to guests*, and illustrate how this property can be used for testing virtual machines.

4.2.1 Virtualisation

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 virtualisation.

We already introduced *CPU emulators* in Section 3.2. *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 virtualised [47]. They identified two special classes of instructions, privileged and sensitive, and derived a sufficient condition under which an ISA can be efficiently (and easily) virtualised. 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 behaviour depends on the configuration of the resources. The ISA is efficiently virtualisable 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 virtualisation. An example is the Intel x86 ISA [50][2]. In order to allow virtualisation in such an architecture all system code of the guest must be analysed to detect *critical instructions*, that is, sensitive, but not privileged, instructions. Any block of code containing a critical instruction

[1] We are (ab)using the term "CPU virtualisation" to explicitly refer to the virtualisation technique also known as *direct native execution* [57].

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

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.

4.2.2 Transparency of Virtual Machines

The ideal CPU emulator and the ideal CPU virtualiser 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 analyse 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 to tell if they are executed in a virtual machine or not. Recalling the formalism introduced in section 3.2.1, we say $\mathcal{C}_E$ is transparent if $\delta_{\mathcal{C}_E}$ is semantically equivalent to $\delta_{\mathcal{C}_P}$.

Clearly, transparency is strictly related to correctness and transparency implies the absence of defects. Our goal is to use the transparency property just defined to analyse 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 virtualiser, lack of transparency could also be caused by intentional design and implementation decisions, made to facilitate virtualisation. For example, a CPU virtualiser could make assumptions about the internals of the guest, or could force the guest into certain configurations that allow to minimise the complexity of the virtualiser and to minimise performance overhead. In any case, the lack of transparency can produce unexpected behaviours in guests.

4.2.3 Testing Transparency of Virtual Machines

CPU emulators and CPU virtualizers, on virtualisation 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 virtualisation friendly ISAs sometimes fail to satisfy this property. Thus, we propose a methodology to test automatically whether such a property is satisfied or not.

The methodology we adopt for testing transparency of Virtual Machines, is very similar to the methodology we adopted in EmuFuzzer. We introduced such methodology in section 3.2.2.

Our approach to test if an emulator $\mathcal{C}_E$, for the CPU $\mathcal{C}_P$, is transparent or not is based on fuzzing [41] and differential analysis [39]. 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 [59]: 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 behaviours of the CPU. Compared to traditional fuzzing, the protocol-specific approach we are using allows to concentrate the efforts mostly on meaningful states.

The problem we address in this work is by far more challenging than the one we considered in Chapter 3. In fact, with EmuFuzzer we focused the testing only on the behaviour of CPU emulators in user-mode. In this work 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.

4.3 KEmuFuzzer

To implement the testing methodology, 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 behaviour 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 behaviour 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 behaviour 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 behaviour 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.

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

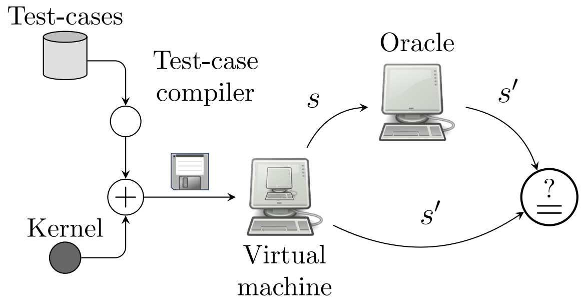


Figure 4.1: *Overview of KEmuFuzzer*

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).

4.3.1 Architecture and Methodology

Figure 4.1 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 initialises the environment for executing the test-case. When the environment is fully initialised, 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.

4.3.2 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 behaviours 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 behaviour of the virtualiser 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 4.1 briefly summarises 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 [19]). 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 4.3.3, 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

Symbolic operator	Description
KEF_INTEGER(n)	Generate a set of n-bit integers
KEF_ITERATE(i_1,...,i_n)	Iterate over i_1,...,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 4.1: *Examples of symbolic operators used in test-case templates*

of which differs in the concrete values returned by the generator functions. Thus, using templates and generator functions we can automatically test the behaviour of the CPU in many different situations.

Figure 4.2 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 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 4.2(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 4.2(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

```
<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 4.2: *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.*

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 KEmuFuzzer's kernel. Finally, Figure 4.2(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 `randomise_ring` attribute of the root node to specify that an execution ring must be randomised (i.e., different test-cases are generated, 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 initialises 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 behaviour 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

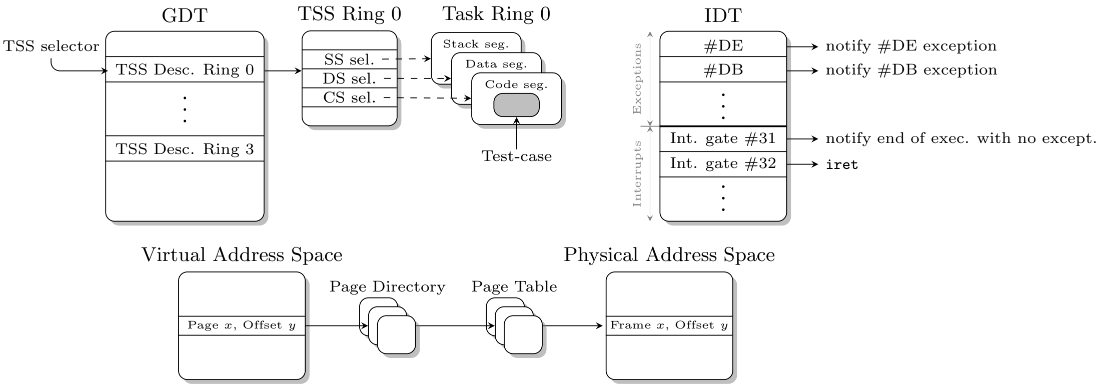


Figure 4.3: *Memory layout of the environment used to execute test-cases*

what happens with the three test-case templates of Figure 4.2. 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.

4.3.3 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 initialising 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 optimised to minimise bootstrap time and to minimise 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 4.3.4 and 4.3.5.

Figure 4.3 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 initialising the Intel x86 CPU in 32-bit protected mode with paging enabled. However, with minimal modifications the kernel can initialise 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 initialises 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 initialised 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 initialises 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 4.3.2). 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 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 initialisation 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 4.3). 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.

4.3.4 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.

4.3.5 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 behaviours 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 virtualisation, namely VT-x [43]. By using hardware-assisted virtualisation, 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 virtualisation-friendly than a ISA that meets Popek and Goldberg minimal requirement for efficient virtualisation. 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 initialise 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 initialising 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 initialise 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) [28], a virtual machine monitor (VMM) for GNU/Linux. KVM architecture is very simple because it runs only on processors with hardware support for virtualisation. 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 initialised and ready to execute a test-case. Roughly speaking, we can create a virtual machine and initialise 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

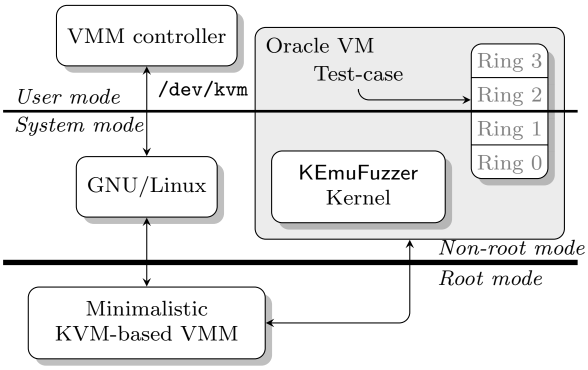


Figure 4.4: *Oracle based on VT-x*

results of our oracle.

Figure 4.4 shows the architecture of our oracle based on hardware-assisted virtualisation. 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 minimises 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.

4.4 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.

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 4.2: *Results of the evaluation*

4.4.1 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 virtualisation since our goal was to test traditional CPU virtualizers that do not leverage such technology.

4.4.2 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 4.2. 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. We re-used the same test-cases generated with the method introduced in Chapter 3 called "CPU-assisted test-case generation" that covers the large majority of the instruction set, while minimising redundancy. Each of the templates generated with this approach executed a different instruction, in system-mode.

4.4.3 Experimental Results

Table 4.2 shows the results of our evaluation. Table 4.3 instead shows the average test-case execution time and the timeout on test-case execution time. For each virtual machine, Table 4.2 reports the number of test-case templates and the number of compiled test-cases for which we observed a non-transparent behaviour. 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 behaviours (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, behaviours to detect if they are executed in a real or in a virtual machine [46, 52].

In the following paragraphs we briefly describe the defects we found in the tested software. In several situations we noticed non-transparent behaviours, 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.

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. 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 behaviour of the `bswap` instruction, when it references a 16-bit register, is also non transparent. The Intel reference states the behaviour of the instruction is undefined when it is executed with 16-bit operands. This is an artifact of our testing methodology and cannot be properly considered a real defect of the emulator.

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.

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

Table 4.3: *Average test-case execution time (timeout on test-case execution time was: 10s*[1] *and 40s*[2]*)*

VirtualBox

VirtualBox handles critical instructions using both emulation and scanning and patching. Scanning and patching is used to virtualise 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.

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. This may seem a very serious defect, however, exceptions must be thrown by the execution of `ret` or `retn` instruction for this bug to be exposed. This can happen if the `stack` is invalid or the instruction pointer taken from the stack points to an invalid location. Fact is real programs hardly behave so bad. We also found that accessed entries of the GDT and accessed entries of the page table are not marked as such.

4.5 Discussion

As we described in Section 4.3.5, the oracle we use in KEmuFuzzer is based on a stripped down version of KVM. During the development, we manually inspected the code of KVM to verify its transparency to guests. Quite surprisingly, we found several defects. As an example, if the execution an I/O instruction raises an exception, KVM still updates the EIP register, while this register is left unchanged by the physical processor. Even if hardware-assisted virtualisation ease the development of CPU virtualizers, the tasks left to the virtual machine monitor are not trivial. All the defects we found in KVM have been confirmed by the developers, and many of them have now been patched in the upstream version.

In the future, we will replace the oracle based on KVM with a minimalistic virtual machine monitor developed from scratch. In this way, we will minimise the possibility of undetected defects in the oracle.

5 Returning to randomised lib(c)

In this chapter we present a new attack to bypass W$\oplus$X and ASLR. The state-of-the-art attack against this combination of protections is based on brute-force, while ours is based on the leakage of sensitive information about the memory layout of the process. Using our attack an attacker can exploit the majority of programs vulnerable to stack-based buffer overflows *surgically*, i.e., in a single attempt. We also propose a new effective protection scheme which does not require recompilation, and introduces only a minimal overhead.

5.1 Introduction

ASLR and W$\oplus$X are two widely adopted protection schemes against threats like memory error exploits, aiming to prevent arbitrary code execution. These protection schemes, combined together, are believed to be effective for avoiding such threats to happen. With W$\oplus$X, code injection isn't possible anymore for data is flagged as not executable. With ASLR, malicious users are deprived of essential information they need to successfully exploit memory error vulnerabilities.

These premises together with the wide adoption of the aforementioned protection techniques by most modern Operating Systems, suggest those schemes provide a good level of security.

It is well-known that randomization is an all-or-nothing solution. If we leave even a single segment unrandomized, then attacks can be easily mounted; little is gained by randomizing some sections and not others [32]. In this chapter we convert theoretical knowledge that such attacks are feasible into a practical one. We show those protection schemes provide only a false-sense-of-security. More precisely, ASLR, if applied partially, can enable exploitation of stack-based buffer overflow vulnerabilities whenever the non-randomised portion of the binary under attack leaks some information on the memory layout of the attacked process.

Current implementation of ASLR in Unix-like systems, leave some sections of elf binary executables not randomised, in particular the code section and the Global Offset Table (GOT) are loaded at a fixed address.

Data present in those sections can be abused to exploit reliably, i.e. in a single shot, programs vulnerable to stack-based buffer overflows. The code section can

```
void sanitize(char *str, int len) {
   char newstr[128];
   int newlen = 0;

   for (int i = 0; i < len; i++) {
      if (str[i] != ...)
         newstr[newlen++] = str[i];
   }
   ...
}
```

Figure 5.1: *Sample vulnerable program*

be abused by recycling existing code through the adoption of techniques like return-oriented programming.

Data present in the GOT, can be used for computing the address of useful functions belonging to the libc, thus enabling an attacker to mount a return-to-libc attack.

This chapter makes the following contributions:

- A new approach to exploit stack-based buffer overflows in programs protected with both W⊕X and ASLR. Our attack is an information leakage attack that exploits information about the random base address at which a library is loaded and can subvert the execution of a vulnerable program and perform a return-to-lib(c) with *surgical precision*, i.e., in a single shot.
- A new protection that is effective at stopping our attack, does not require recompilation of any executable, and introduces only negligible overhead.

5.2 Background

Figure 5.1 shows a sample program vulnerable to a traditional stack-based overflow. An attacker can exploit the buffer overflow to overwrite the stack with arbitrary data, thus forcing the program to execute arbitrary code. Modern operating systems mitigate this class of attacks with W⊕X, a policy that prevents memory pages containing executable data from being writable and vice versa. With such a policy in place, the only way for an attacker to execute arbitrary code is to mount a return-to-lib(c) attack [14], which consists of overwriting the return address of the vulnerable function and the following words of the stack

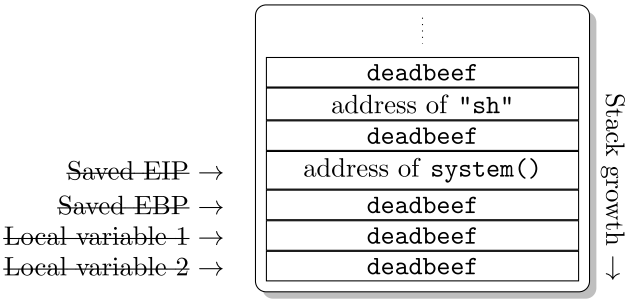


Figure 5.2: *Stack of the vulnerable process during a* `return-to-libc` *attack*

with the address of a lib(c) function (e.g., `system`) and the arguments to pass to this function (e.g., the address of the string `"sh"`). Figure 5.2 shows the stack of the vulnerable process after the overflow, prepared to mount the return-to-lib(c) attack.

When address-space layout randomisation (ASLR) is used in tandem with W⊕X, the attack becomes much more difficult. At every execution, the stack, the heap, and shared libraries (such as libc), are loaded at different random addresses. Consequently the attacker does not know the address of the function to return to. Nevertheless, Shacham *et al.* demonstrated that return-to-lib(c) attacks are still feasible in systems protected with address- space layout randomisation using brute force [55]. The expected number of attempts for the brute force attack to succeed is 2^n, where n is the number of bits of randomness in the address-space. As an example, on GNU/Linux on IA-32 (where n is at most 16) 65,536 attempts are sufficient to successfully mount the attack.

5.3 Attack

5.3.1 Overview of the attack

Our attack against address-space layout randomisation successfully returns to lib(c) in a single attempt, while Shacham *et al.*'s attack instead requires 2^n attempts (where n is the number of bits of the address-space subject to randomisation). We propose an information leakage attack. Contrarily to the information leakage attack suggested by Durden [16], ours requires neither information about the current layout of the process, nor the ability to access arbitrary stack elements (e.g., to retrieve the address of `main()`). Instead, our attack exploits information about the base address of the lib(c), which is directly available in the memory of the process. The attack is built on an exploit technique [27, 54], that was previously thought to be inapplicable with ASLR. Our idea is to combine few

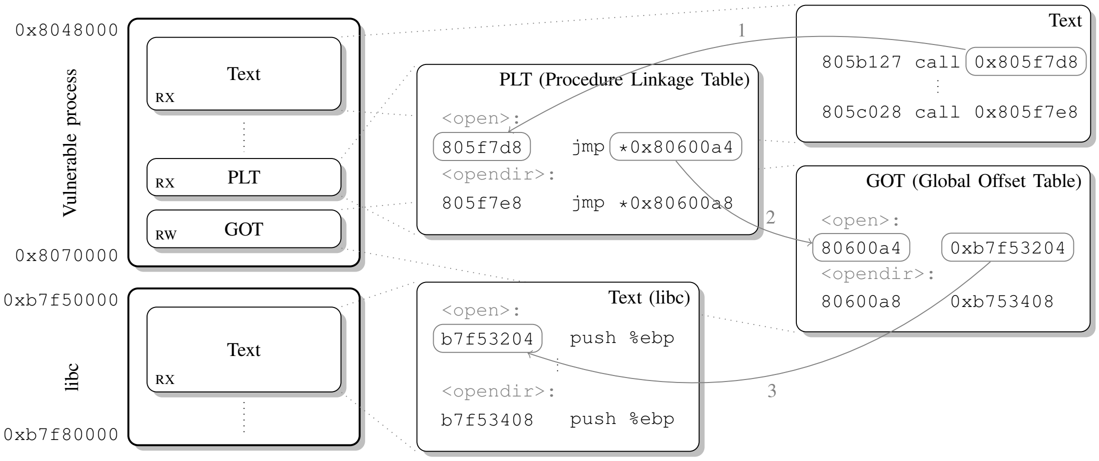
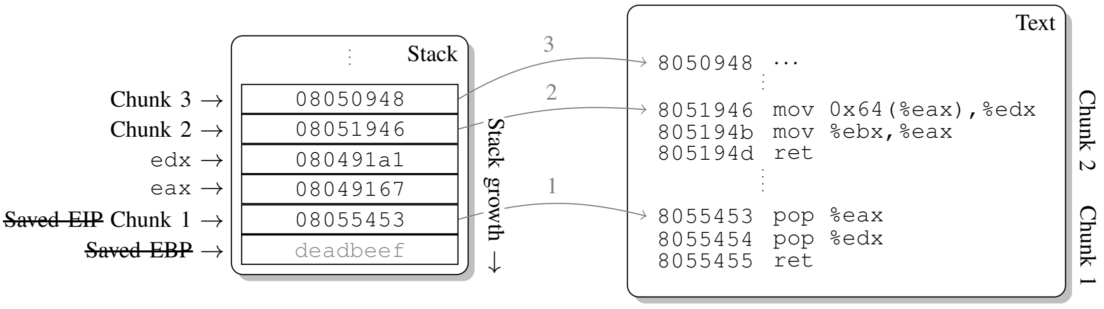


Figure 5.3: *Layout of a sample process and overview of the mechanism used to invoke functions residing in shared libraries*

code fragments that, despite ASLR, are available at absolute fixed addresses in the memory of the vulnerable process and to use these fragments to discover the base address of a dynamic library. Once the library has been de-randomised, we can return to any of its functions.

Figure 5.3 shows the layout in memory of our sample vulnerable program[1]. To ease the presentation, the layout is simplified: the stack and the heap are omitted and we assume that dynamic binding between the executable and the shared library has already been performed. The vulnerable program is loaded at address `0x8048000`. We assume the vulnerable program is compiled to be position dependent (that is the default compiler configuration) and consequently that its base address is fixed. This assumption is well-grounded because supported by empirical evidence: the large majority of executables found in modern UNIX distributions are not position independent (about 92.9%), only shared libraries are. The lib(c) instead is loaded at address `0xb7f50000`, but the address varies from execution to execution. The figure also shows the mechanism used to invoke the functions exported by the shared library, which is essentially an indirect jump table [31]. When the executable is loaded in memory, the linker transfers in memory all the requested libraries and then performs the relocation. The executable contains two special data structures (or sections) used specifically for the purpose of linking the executable with shared objects: the Global Offset Table (GOT) and the Procedure Linkage Table (PLT). The former is an array containing the address of the various library functions used by the program. The latter is an array of jump stubs. The i^{th} PLT entry contains a jump instruction

[1] All the examples are specific for the x86 architecture, GNU/Linux (2.6.x), and ELF-32 executables. We use the AT&T assembly syntax.



```
8050948 ···
   ⋮
8051946 mov 0x64(%eax),%edx
805194b mov %ebx,%eax
805194d ret
   ⋮
8055453 pop %eax
8055454 pop %edx
8055455 ret
```

Figure 5.4: *Sample stack configuration with three gadgets, to chain the code chunks available in the vulnerable process*

that jumps to the address stored in the i^{th} GOT entry. The linker, at load time (assuming preemptive binding), fills the GOT with the addresses of the imported functions, updated to be consistent with the current base address of the library. The separation between PLT and GOT is for improved security: the former is executable but not writable, the latter is writable but not executable, thus preventing an attacker from writing and executing arbitrary code. For example, to invoke the libc function `open` our sample program performs a function call (instruction `0x805b127`), but instead of invoking a normal function, it invokes the stub for `open` in the PLT (located at address `0x805f7d8`). In turn, the stub of the PLT jumps to the code of the function inside the libc. The jump is indirect and the target of the jump is the address stored in the GOT entry of the `open` function (at address `0x80600a4`). In summary, through the call and the indirect jump the execution flows to `open` in the libc (in our sample process, the absolute address of `open` is `0xb7f53204`).

The knowledge of the absolute address of a single function exported by the lib(c) is sufficient to mount a successful attack, enabling any function in the library (including those not exported) to be invoked. Our attack exploits the information found in the GOT of the process to calculate the base address of the library, calculate the absolute address of an arbitrary function of the library, and subsequently invoke that function. Let *offset*(s) be a function that computes the virtual offset, relative to the base address of the library, of the symbol s. It is worth noting that the virtual offset can be computed off-line from the library file and that the offset is constant. To ease the presentation, we use `open` to denote any function used by the attacker to de-randomise the library, and `system` to denote any function whose absolute address the attacker wants to compute. Given the absolute address of a library function, the base address of the library (`libc`) can be computed as follows:

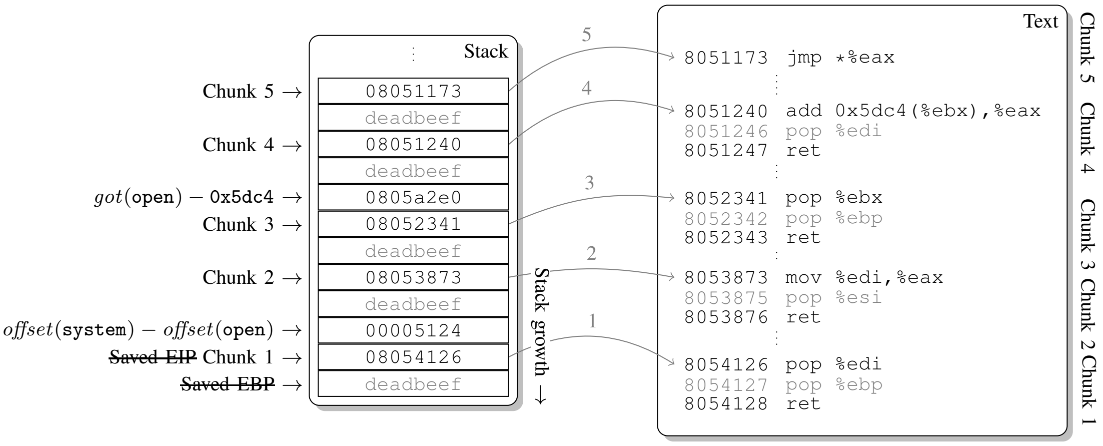
$$\texttt{libc} \;=\; \texttt{open} - \mathit{offset}(\texttt{open})$$

Similarly, the absolute address of any function of the library can be computed as follows:

$$\texttt{system} \;=\; \texttt{open} - \mathit{offset}(\texttt{open}) + \mathit{offset}(\texttt{system})$$

Even though the math is trivial, it is very complex to perform in our context. Indeed, despite the stack overflow vulnerability, we cannot inject and execute our own code because the stack and all other data pages are not executable. A solution to overcome this limitation is to *borrow code chunks*, that is, to use code already available in the executable section of the process [27, 54]. Practically speaking, a code chunk is a sequence of bytes representing a sequence of one or more valid instructions that is terminated by a `ret` instruction. Although code chunks available are typically very simple and short, they can be combined, using *return-oriented programming*, by constructing powerful *gadgets*, i.e., short blocks placed on the stack that chain several code chunks together and that perform a predetermined computation [54]. An example of a code chunk is the string `8b 50 64 c3`, corresponding to the sequence of instructions `mov 0x64(%eax),%edx; ret`. The `ret` instruction ending each code chunk allows the construction of gadgets that link multiple chunks together. Figure 5.4 shows a sample stack configuration containing two gadgets that combine a 3-byte code chunk (the sequence of instructions `pop %eax; pop %edx; ret`) with another one, to read the content of arbitrary memory locations. During the overflow, the stack frames of the vulnerable function and the callers are overwritten with gadgets (see Figure 5.4). The first gadget starts exactly where the return address of the vulnerable function was stored before the overflow. It is composed of three double-words: the address of the first code chunk (`0x08055453`), and two integers (`0x8049167` and `0x80491a1`) that will be consumed during the execution of the code chunk. When the vulnerable function returns, the first code chunk is executed, and its execution results in the initialisation of the two registers (i.e., `eax` and `edx`) with the values specified in the gadget (the second and third double-words of the gadget). The second gadget, being stored adjacently to the first one, causes the execution to flow from the first to the second code chunk. Indeed the `ret` instruction terminating the first code chunk references the double-word belonging to the subsequent gadget and representing the start address of the second chunk (`0x08051946`). The second code chunk reads the content of the memory location pointed by `eax` and stores the result in `edx`. Additional operations could be chained to perform more complex computations by writing other gadgets to the stack during the overflow.

The x86 architecture has a very dense and rich instruction set, instructions have variable length and do not need to be aligned. Therefore, code chunks are typically very frequent. However, those usable by an attacker are just a few. The numerous code chunks available in libc and other libraries cannot be used because



```
8051173  jmp *%eax
   ⋮
8051240  add 0x5dc4(%ebx),%eax
8051246  pop %edi
8051247  ret
   ⋮
8052341  pop %ebx
8052342  pop %ebp
8052343  ret
   ⋮
8053873  mov %edi,%eax
8053875  pop %esi
8053876  ret
   ⋮
8054126  pop %edi
8054127  pop %ebp
8054128  ret
```

Figure 5.5: *Sample stack configuration for the GOT dereferencing attack, where the address of* `system` *is assumed to be* `0xb7f58328` *(instructions and elements of the stack irrelevant for the attack are shaded).*

of ASLR. As the executable is position dependent, only a few constant-address chunks in the code section can be used.

5.3.2 Details of the attack

Our attack uses the code chunks available in the code section of the vulnerable process to determine the base address of the lib(c), and uses this information to execute any function of the library. More precisely, our attack works as follows.

1. Identify the code chunks available in the vulnerable process.

2. Combine these code chunks to retrieve from the GOT of the vulnerable process the absolute address of a function of the lib(c).

3. Compute, again using the available code chunks, the absolute address of the function of the library we want to invoke.

4. Transfer the control of the execution to the latter function.

We present two variants of the attack. The first one is a straightforward application of the four steps described above. The second one has been developed to operate in situations where the first variant cannot, because the required code chunks are not available. The second variant indeed uses more common code chunks that allow to modify any entry of the GOT, without reading it explicitly.

Attack 1 – GOT dereferencing

The first attack combines gadgets to read the absolute address of any lib(c) function (e.g., `open`) from the GOT of the process, uses this address to compute the absolute address of another function of the library (e.g., `system`), and jumps to the address just computed. To do that we need the following gadgets: a load, an addition, and an indirect control transfer. Each of these gadgets can be obtained by combining one or more code chunks available in the code section of the vulnerable program. The x86 architecture facilitates the attack because it can perform complex tasks, such as a load and an arithmetic operation, with a single instruction. Therefore, the number of code chunks required to mount the attack is very small.

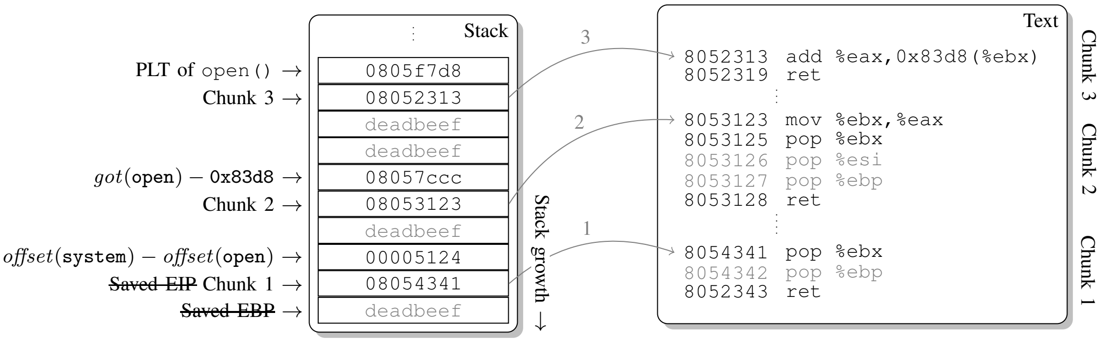
An example of a code chunk that constitutes one of the building blocks of our attack is the sequence of bytes `03 83 c4 5d 00 00 5f c3`, encoding the instructions `add 0x5dc4(%ebx),%eax; pop %edi; ret`. To turn such a code chunk into a dangerous gadget, it is sufficient to properly initialise the registers `eax` and `ebx`. Indeed, a proper configuration of the two registers enables to load the absolute address of `open`, and to compute the address of `system`. Let $got(s)$ be the address of the GOT entry of the symbol s. Like for the virtual offset of a symbol, the addresses of the various GOT entries of the program are constant, and can be computed off-line from the program file. The assignment to the two registers necessary to compute the absolute address of `system` is:

$$\begin{aligned}\texttt{eax} &= \mathit{offset}(\texttt{system}) - \mathit{offset}(\texttt{open}) \\ \texttt{ebx} &= \mathit{got}(\texttt{open}) - \texttt{0x5dc4}\end{aligned}$$

With this register configuration the instruction loads from the GOT the absolute address of `open` (the $-$`0x5dc4` delta is necessary because the instruction loads the data at address `ebx + 0x5dc4`) and sums it to the offset stored in `eax`. The result is saved in `eax`, and corresponds to the absolute address of `system`. To complete the attack, the attacker just needs a code chunk that transfers the execution to the address in `eax`. For example, the instruction `jmp *%eax` can be used for this purpose.

Figure 5.5 shows the stack of the sample vulnerable process during the attack and illustrates how the various code chunks are combined in gadgets by the attacker to perform the exploit. Overall, during the attack, the stack contains five different gadgets. The number can vary slightly, depending on the type of code chunks available in the vulnerable program[2]. The first code chunk (at address `0x8054126`) pops two double-words from the stack and stores them in

[2] The gadgets used to illustrate the attack resembles the ones more common on GNU/Linux (x86) systems.



```
8052313  add %eax,0x83d8(%ebx)
8052319  ret
            ⋮
8053123  mov %ebx,%eax
8053125  pop %ebx
8053126  pop %esi
8053127  pop %ebp
8053128  ret
            ⋮
8054341  pop %ebx
8054342  pop %ebp
8052343  ret
```

Figure 5.6: *Sample stack configuration for the GOT overwriting attack*

edi and ebp respectively. The attacker uses the first gadget to initialize edi with the distance between system and open. The register ebp is irrelevant for the attack and its initialisation is just a side effect of the code chunk. Indeed, the code chunk resembles a standard function epilogue, which restores callee saved registers. The ret instruction terminating the first code chunk triggers the second gadget, stored in the stack right above the element previously popped into ebp. The gadget uses the second code chunk (at address 0x8053873) to copy the value of edi to eax. This operation is needed because we are assuming that no code chunks exists to directly initialise eax. Again, the pop instruction found in the chunk is a side effect. The third chunk (at address 0x8052341) is used by the attacker to initialise ebx with the address of the GOT entry of open. The code fragment pops the value from the stack and saves in ebx. After the execution of the first three gadgets both eax and ebx are initialised as described earlier and the attacker has completed the preparation of the context for the execution of the gadget that computes the desired absolute address of system. The fourth gadget is used for the computation and to store the address in eax, and the fifth gadget is used to jump at the beginning of the system function, completing the attack.

Attack 2 – GOT overwriting

The second attack overwrites an entry of the GOT (e.g., the entry of open) with the address of another library function (e.g., the address of system), and transfers control to the selected function through the modified GOT entry. The attack is possible because, in the default setup, binding is performed lazily, and the GOT must be filled on demand. Hence, it must be writable.

The attacker needs the following gadgets: a load, an addition, a store, and an indirect control transfer (with a memory operand). Although apparently more gadgets are needed to perform this variant of the attack than to perform the previous one, in practice the first three operations can be performed using a single machine instruction; that is, an arithmetic operation with a destination memory operand, such as `add %eax,0x83d8(%ebx)`. This kind of code chunk is increasingly frequent in executables, relative to the type of chunk on which the GOT dereferencing attack is based. Furthermore, no particular control transfer instruction is requested to invoke the chosen library function as the PLT stub of the function whose GOT entry has been modified can be used for the attacker's purpose.

Figure 5.6 shows the stack of the sample vulnerable process during the attack, and illustrates how the various code chunks are combined in gadgets by the attacker to exploit the vulnerability. In total the attacker combines three gadgets, two of which perform two operations instead of a single one. The return address of the vulnerable function is overwritten with the address of the first gadget and the previous double-word in the stack contains the distance between `system` and `open`. The first gadget (using the code chunk at address `0x8054341`) initialises the value of `ebx` with the distance stored in the stack. The second gadget (using the chunk at `0x8053123`) copies the value from `ebx` to `eax` and then initialises `ebx` with the address of the GOT entry of `open` that will be overwritten. The third gadget (using the chunk at `0x8052313`) computes the absolute address of `system`, as in the first attack, but with the operands in the inverse order, such that the computed address is stored directly in the dereferenced entry of the GOT. Finally, the `ret` instruction of the gadget is used to return directly to the PLT entry of `open`, in order to use its jump stub to invoke the function through the GOT.

5.4 Attack mitigation

This section presents various protections mechanisms proposed in literature, and discusses their effectiveness at preventing our attack, when used in combination with W⊕X and ASLR. Furthermore, this section presents a new protection that can be used to block both variants of our attack.

5.4.1 Preventing unsafe accesses to GOT

Our attack is not possible on position independent executables (i.e. PIE). However, this feature is not yet widely adopted by modern UNIX distribution, but the motivations for such a choice are not clear (numbers are given in Section 5.5). We speculate that vendors are afraid of the performance penalties PIE could introduce and are also not aware of its real importance. Although we strongly encourage vendors to move to position independent executables, we propose a new

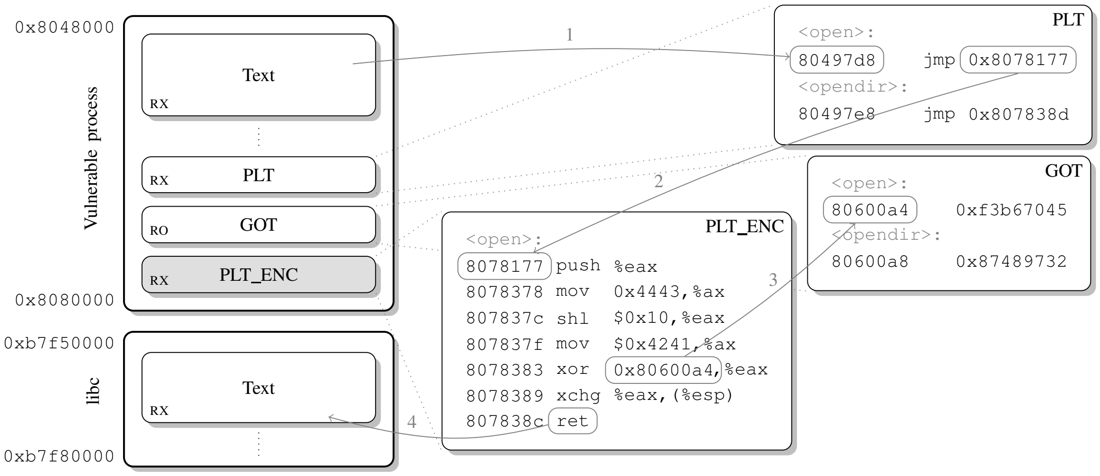
	GOT deref.	*GOT overw.*	*Requires recomp.*
W⊕X and ASLR	–	–	No
Periodic re-randomization [5]	–	–	Yes
GOT randomisation [65]	–	–	No
GOT read-only [19]	–	✓	No
PIE [62]	✓	✓	Yes
Self-randomisation [6]	✓	✓	Yes
Encrypted GOT	✓	✓	No

Table 5.1: *Comparison of existing protections with respect to our attack and to the new proposed protection (✓ denotes that the defence technique prevents the attack)*

runtime solution that, being applicable without recompilation, can be used during the transition to PIE-enabled distributions and on operating systems where PIE is not yet available, but ASLR is (e.g., OpenBSD).

Our solution is inspired by the randomised GOT protection proposed by Xu *et al.* [65], and relies on *encrypting the content of the GOT*. The idea is to encrypt GOT entries, to prevent all but legitimate accesses. With the exception of the accesses performed by the dynamic linker to bind the executable with the shared libraries, all further accesses to the GOT are reads and occur only from the PLT (see Figure 5.3). Therefore, besides the linker, only the accesses originating from the PLT should be considered legitimate and authorised to access to unencrypted content of the GOT and to transfer the execution to the functions in shared libraries. To ease the presentation we assume preemptive binding (i.e., `LD_BIND_NOW` is set). In such a situation all legitimate accesses to the GOT originate from the PLT. However, the approach we are proposing could be extended to work also with lazy binding, by customising the dynamic linker.

In more detail, our scheme operates as follows. We encrypt all the entries of the GOT such that attacker's attempts to read the content of the GOT to guess the random base address of the library fail; without the decryption key, the retrieved content of the GOT is meaningless. Similarly, attempts to modify the GOT fail as well. Obviously, encryption interferes with the correct execution of the program. For this reason, we rewrite the program to make it able to decrypt the protected data when it legitimately accesses the GOT. As all legitimate accesses go through the PLT, it is sufficient to patch each stub of the PLT to dereference and decrypt the corresponding GOT entry, and then to transfer the execution to the decrypted address. The weakness of the randomised GOT protection proposed by Xu *et al.* is that the PLT leaks the address of the GOT, and consequently an attacker can mount both a GOT dereferencing and GOT overwriting attacks. As we adopt a



```
<open>:
8078177 push %eax
8078378 mov  0x4443,%ax
807837c shl  $0x10,%eax
807837f mov  $0x4241,%ax
8078383 xor  0x80600a4,%eax
8078389 xchg %eax,(%esp)
807838c ret
```

Figure 5.7: *Layout of the sample vulnerable process with our of GOT encryption protection enabled*

similar strategy, we are exposed to the same risk. Therefore, we have to protect the patched PLT to avoid any information leak that can be exploited by the attacker.

Each PLT entry is patched to perform the following operations: (I) to read the corresponding GOT entry, (II) to decrypt the address read, (III) and to jump to the decrypted address. Because of the aforementioned problem, the decryption key cannot be stored directly in the code of the patched jump stub, nor can it be referenced explicitly from the code. The solution we adopt inlines in the i^{th} jump stub a key generation function that computes dynamically the decryption key to use for the decryption of the i^{th} entry of the GOT. Encryption keys and key generation functions are generated at runtime and differ from one GOT/PLT entry to another. The rationale behind this choice is that, although an attacker could read (using our GOT dereferencing attack) the code of the patched jump stub that performs the decryption, and try to "borrow" the decryption code, he does not know how to use this code. Indeed, this code is generated randomly at each execution, and to construct useful gadgets from it the attacker would have to analyse (i.e., disassemble) the code, and the only way to do that is to use other gadgets. Although that is theoretically possible, such a complex analysis requires an arsenal of gadgets that are practically impossible to construct even from a large executable.

We have developed a prototype implementation of the proposed protection, for GNU/Linux (x86). For simplicity the prototype requires preemptive binding of shared libraries and does not support dynamic loading of shared objects (e.g., `dlopen`). However, the dynamic linker could be extended to support our protection also with lazy binding and dynamic library loading. Lazy binding typically introduces less overhead and reduces startup costs, and consequently the overhead introduced by our protection could be reduced by completing the prototype. Our prototype consists of a shared library that is injected in the address space of the program to protect (using `LD_PRELOAD`). The library encrypts all the entries of the GOT and then patches the jump stubs of the PLT as described above. Since the size of PLT entries is insufficient to hold our patched code and cannot be enlarged without breaking the functionality of the program, we allocate a new executable section, and store in it the new patched entries. Additionally, we update PLT entries to redirect the execution to the corresponding entries in PLT_ENC. Keys generation functions are constituted of a random number (up to a dozen) of different assembly instructions, and are crafted to be unpredictable. Figure 5.7 shows the memory layout of our sample process with the randomised GOT protection in action. The extra section called PLT_ENC is the section created to hold the new encryption-aware jump stubs. When the program calls the `open` function, the execution flows, through the patched PLT, to the jump stub of `open` in the PLT_ENC section (at address `0x08078177`). The code we use to decrypt the GOT entry of `open` and then to invoke the function looks like the code in the figure, but it is different in each entry and execution. The code performs the three operations necessary to invoke the function (load, decryption, and control transfer) and takes the precaution of preserving all registers. Decryption keys are reconstructed on-the-fly from data embedded in the PLT_ENC stub.

5.5 Evaluation

We evaluated the proposed attack and solution. Overall, the evaluation demonstrated the wide-scale applicability of our attack, and the effectiveness of the proposed protection. Details of the evaluation are reported separately in the following sections.

5.5.1 Evaluation of the attack

We performed two independent evaluations for our attack. First, we tested our attack against a version of Ghostscript vulnerable to a stack-based overflow. We successfully exploited the vulnerable program with both variations of our attack. Second, we tested a large corpus of programs, collected from different UNIX distributions for the x86 and x86-64 architectures and supporting both W⊕X and ASLR, to measure how many of them were *predisposed to the attack* (i.e., whether the attack would be possible if the programs were vulnerable to a stack-based buffer overflow). For the x86 architecture, the majority of the programs tested, about 95.6%, were found to be predisposed to the attack. For the x86-64 architecture we found less predisposed programs, only about 61.8%. This is due to the fact that on x86-64 instructions with 64-bit operands requires a special prefix, and consequently the code chunks needed for the attack are less frequent.

Automation of the attack

For the evaluation we have implemented a prototype tool, called SARATOGA, that automatically analyses a ELF-32 or ELF-64 executable (for x86 and x86-64 architectures respectively), detects whether the program is predisposed to any of the two variations of the attack, and generates a stack configuration that can be used to exploit a vulnerability in the program. To find code chunks in an executable, SARATOGA uses the algorithm presented by Shacham *et al.* [54]. SARATOGA combines available code chunks using a custom algorithm we have developed. Our algorithm is goal-oriented and rule-based. Given a code chunk that either allows to dereference or to overwrite a GOT entry, the algorithm assigns a predetermined value to each possible use of the code chunk (e.g., the source operands of the instructions in the chunk). The algorithm uses a set of combination rules and tries to apply them recursively, to combine the available code chunks to perform the requested assignments. If multiple combinations are possible, the algorithm selects the one consuming less stack space. For output, SARATOGA produces a stack configuration containing the gadgets for the exploitation.

Exploiting a real vulnerability

We tested our attack against a vulnerable version of Ghostscript [12]. We initially developed a conventional exploit and tested it against the program with W⊕X and ASLR disabled. The exploit worked correctly and gave us a shell. Subsequently, we enabled the two protections and verified that the exploit stopped working. We run SARATOGA on the image of the program under attack and, in few seconds, obtained two stack configurations for the two variants of the attack. We constructed two exploits using the results provided by SARATOGA, and successfully exploited the vulnerability and obtained a shell with both.

Wide-scale applicability of the attack

The evaluation targeted the executables found in the directories `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` of the following distributions: GNU/Debian “Squeeze” (x86), GNU/Debian “Lenny” (x86-64), Fedora “Cambridge” (x86), OpenBSD 4.5 (x86-64). For the total set of executables found in each distributions we selected for the evaluation only those whose code size was greater than 20Kb. The rationale was that excessively small executables have a limited functionality and

	Deb. (x86)	*Deb. (x86-64)*	*Fedora (x86)*	*O.BSD (x86-64)*
Executables	509	333	590	174
non-PIE	95.7%	97.3%	85.8%	100%
Writable GOT	99.8%	100%	99.0%	100%
Attack 1	64.0%	17.8%	49.5%	58.6%
Attack 2	96.1%	57.4%	95.0%	68.4%
Any attack	96.3%	58.3%	95.0%	68.4%

Table 5.2: *Experimental evaluation of the effectiveness of the attack on x86 and x86-64 executables*

	`bc`	`bogof.`	`bzip2`	`clamscan`	
PIE	10.55%	3.46%	0%	.12%	
Enc.GOT	.21%	15.49%	.63%	.11%	
	`convert`	`grep`	`oggenc`	`tar`	*Avg.*
PIE	0%	1.41%	.16%	.12%	1.98%
Enc.GOT	.32%	4.54%	.02%	.20%	2.69%

Table 5.3: *Overhead introduced by PIE and by our protection (the baseline for comparison are the non-PIE executables)*

very seldom attract attackers. On the contrary, commonly-attacked executables (e.g., Ghostscript, Samba, Apache) are bigger, of the order of tens or hundreds of kilobytes.

The results of our evaluation are reported in Table 5.2. For each distribution, the table reports the total number of executables analysed, the percentage of position dependent executables, the percentage of executables with writable GOT, the percentage of executables vulnerable to the GOT dereferencing attack, the percentage of executables vulnerable to the GOT overwriting attack, and the percentage of executables vulnerable to any of the two attacks. All three tested Linux distributions support PIE and non-writable GOT. Unfortunately, our results testify that these mitigation techniques are not yet widely used. As the table shows, the large majority of the executables for x86 are predisposed to at least one of the two variants of the attack. The second variant has much larger applicability, because the requested code chunks are more common. The attack is not as effective on x86-64 executables, but still, more than half of the tested executables are predisposed to it. With the exception of OpenBSD executables (where PIE is not available), all the executables found in the other distribution would not be predisposed to the attack if they were PIE. Furthermore, considering that the number of programs predisposed to the GOT dereferencing attack is much smaller that the percentage of programs predisposed to at least one of the two attacks, the read-only GOT protection would give non-negligible benefits.

It is worth noting that the a vulnerability found in an executable predisposed to our attack might not be exploitable. For example, the vulnerability might not expose a large enough portion of the stack, or it might not provide the needed stack manipulation operations (e.g., to inject null bytes). These situations are not considered a limitation of our attack but rather a limitation of the vulnerability itself.

5.5.2 Evaluation of the proposed defence

We evaluated the efficacy our encrypted GOT protection, as well as the overhead it imposes. Our results demonstrate the effectiveness of our solution at stopping both attack variations, as with a small runtime overhead (about 2.69%).

To evaluate the effectiveness of the proposed mitigation strategy we tested the two exploits constructed with the help of our tool against the vulnerable version of Ghostscript, with our GOT protection, W⊕X, and ASLR enabled. Both exploits failed to work. The vulnerable process terminated with a page fault exception caused by an access to an invalid memory page.

We evaluated the overhead introduced by our protection and compared with the overhead introduced by PIE. For the evaluation we used the following applications: `bc`, `bogofilter`, `bzip2`, `clamscan` , `convert`, `grep`, `oggenc`, and `tar`. These applications are CPU-bound and make frequent use of functions in shared libraries. Experiments were performed on an x86 system running GNU/Linux 2.6.27. As our protection works entirely in user-space, for each application we measured the user-time requested to complete a batch job, averaged over multiple runs, in three different configurations: (I) with a version of the executable compiled with default options (position dependent executables), (II) with a version of the executable complied with the default options as PIE, and (III) with the first version of the executable but with our runtime protection enabled. Table 5.3 reports the overhead measured with each application and the average. The percentages in the table represent the relative increment of user-time, with respect to configuration (I). From these results we can draw two main conclusions. First, the average overhead introduced by PIE is very small, 1.98%, and a maximum of 10.55% with `bc`, and can be further reduced with more aggressive compilers optimisations (e.g., by omitting the frame pointer). Second, the overhead introduced by our encrypted GOT protection is also very small and comparable to that introduced by PIE. The average overhead observed was 2.69% and a maximum of 15.49% with `bogofilter`, which invokes library functions with a very high frequency. The small overhead implies practical adoption of our protection on both end-users and production systems.

5.6 Discussion

The proposed protection scheme is clearly focused only on information leakage located in the GOT. We insist saying PIE should be considered the correct solution to attack techniques presented in this dissertation. Our attack techniques allow an attacker to compute and jump to a particular libc function not originally used by the binary itself. The defense scheme we proposed prevents such computation but cannot do anything against the re-use of an already linked function. Practically speaking, there is no point in computing `system`'s address if the vulnerable binary already uses it. An attacker could simply jump to `system`'s PLT stub to gain control of the vulnerable process. Moreover, our protection scheme does not prevent code-reuse, so attacks are still possible, provided the vulnerable binary has enough gadgets.

6 Conclusions and future work

Emulators are designed without transparency in mind, having the goal of being able to execute as best as they can typical application software. Application software which does not rely on effects declared undefined by hardware vendors. Emulators are actively used for dynamic behavioural analysis of suspicious programs by researchers. Besides, malware writers, assuming the presence of an emulator likely means their software probably is under analysis, do rely on those undefined effects. More generally they rely on behavioural differences between real and emulated hardware for hiding their malicious behaviour and stay undetected for a longer time. This is true even for system virtual machines based on native code execution, as they have to trap and emulate a subset of the instruction set (sensitive instructions). This dissertation provided ways to detect behavioural differences between the real hardware and the emulated one. The results from the proposed testing methodologies can then be used to harden emulators and system virtual machines to the point they can be considered transparent, i.e. the transition function implemented by the real hardware and by the emulator or the virtual machine are exactly the same.

Advances in defensive technologies are constantly raising the bar against the exploitation of vulnerabilities, which are one of the most effective threat vectors used by malware for their large-scale diffusion. As defensive technologies are improving, attack techniques become more sophisticated finding new paths to reach the same goal: obtaining complete control of the vulnerable software. This dissertation provided evidences of the exploitability of software vulnerable to stack-based buffer overflows even if they are "*protected*" by state-of-the-art defence techniques, thus exposing their defects and providing also some improvements to them (where the PIE solution is not an option).

In the following we briefly suggest some direction for future research in each involved topic.

Chapters 3 and 4 Some enhancements are possible on the proposed testing methodologies. More precisely, the proposed technique is based on the assumption made on the virtual machine monitor used as an oracle for the implementation of KEmuFuzzer: the oracle behaves exactly like the real hardware. During the experiments, some bugs were found on the KVM code, bugs affecting the virtualised environment. This means there is room for guest-detectable misbehaviours even with hardware-assisted virtualisation. The ideal oracle would be the real hardware running in its native execution modes.

Chapter 5 ASLR applied to the entire process space seems to solve the problem posed by memory error vulnerabilities leading to arbitrary computations by attackers. With PIE enabled binaries, attackers can no more build a stack layout with code pointers pointing to attacker-supplied code. Neither they can point to code already present in the process' address-space. At least they cannot do that in a reliable way. With a 32-bit address-space they can successfully exploit such vulnerabilities by means of brute-force. In a 64-bit address space this is nearly impossible. Nevertheless, talking about stack-based buffer overflows, it is still possible to partially overwrite the first encountered return address and be able to reliably jump to already present code in a relative fashion. Taking control of a vulnerable process by overwriting one single code pointer seems unlikely to be possible, anyway we think this can be a research direction worth to be taken.

Bibliography

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

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

[3] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research Annual Conference (EICAR 2006)*, 2006.

[4] Fabrice 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.

[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[6] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[7] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn 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.

[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. In *Proceedings of the 3rd International Workshop on Information Assurance*, pages 434–441, 2007.

[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA, 2008. ACM.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[11] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[12] CVE-2008-0411. Ghostscript zseticcspace() Function Buffer Overflow Vulnerability.

[13] Brett Daniel, Danny Dig, Kely Garcia, and Darko 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, September 2007.

[14] Solar Designer. ”return-to-libc” attack. Bugtraq, 1997.

[15] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[16] Tyler Durden. Bypassing PaX ASLR protection, July 2002.

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

[18] Tal Garfinkel and Mendel 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, February 2003.

[19] GNU `binutils`. http://www.gnu.org/software/binutils/.

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

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

[22] Alex Groce, Gerard Holzmann, and Rajeev 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.

[23] grsecurity.

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

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

[26] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riles Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* John Wiley & Sons, 2004.

[27] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005.

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

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

[30] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, September 1996.

[31] John Levine. *Linkers and Loaders.* Morgan Kaufmann, 2000.

[32] Lixin Li, James E. Just, and R. Sekar. Address-space randomization for windows systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 329–338, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Henry A. Lichstein. When Should You Emulate? *Datamation*, 1969.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim 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.

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

[36] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*. ACM, July 2009.

[37] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 171–182, New York, NY, USA, 2010. ACM.

[38] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John 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, September 2008.

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

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

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

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

[43] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[44] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.

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

[46] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo 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, August 2009.

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

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

[49] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting System Emulators. In *Proceedings of Information Security Conference (ISC 2007)*. Springer-Verlag, 2007.

[50] John Scott Robin and Cynthia 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.

[51] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society.

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

[53] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference*, 2005.

[54] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, October 2007.

[55] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

[56] Emin Gün Sirer and Brian N. Bershad. Testing java virtual machines. In *Proceedings of the International Conference on Software Testing And Review*, nov 1999.

[57] Jim E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

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

[59] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[60] The PaX Team. Address space layout randomization.

[61] The PaX Team. PaX non-executable pages.

[62] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3, August 2004.

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

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

[65] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent Runtime Randomization for Security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.