RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

Technical Report HGI-TR-2010-001

ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks

Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy

System Security Lab
Ruhr University Bochum, Germany

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

HGI-TR-2010-001
First Revision: March 19, 2010
Last Update: December 14, 2010

ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks*

Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy

Abstract

Return-oriented programming (ROP) is a technique that enables an adversary to construct malicious programs with the desired behavior by combining short instruction sequences that already reside in the memory space of a program. ROP attacks have already been demonstrated on various processor architectures ranging from PCs to smartphones and special-purpose systems.

In this paper, we present our tool, ROPdefender, that dynamically detects conventional ROP attacks (that are based on return instructions) with a reasonable runtime overhead of 2x. In contrast to existing solutions, (i) *ROPdefender* does not rely on *side information* (e.g., source code or debugging information) and (ii) it instruments all return instructions issued during program execution including all returns from dynamic libraries, even if the adversary subverts the control-flow by other means. Moreover, *ROPdefender* can handle Unix signals, non-local control transfers, C++ exceptions, lazy binding, and can be applied to multi-threaded applications such as Mozilla Firefox or Acrobat Reader. Finally our implementation supports mainstream operating systems (Windows and Linux) for the Intel x86 architecture. As proof of concept we show that *ROPdefender* successfully detects recent Acrobat Reader exploits on Windows.

1 Introduction

Runtime attacks on software aim at subverting the execution flow of a program by redirecting execution to malicious code injected by the adversary. Most of these attacks are memory-related and typically exploit a buffer overflow vulnerability on the stack [4] or the heap [5]. It seems that buffer overflows are still the dominant vulnerability in today's applications: According to the NIST[1] Vulnerability database and as depicted in Figure 1, the number of reported buffer overflow vulnerabilities continue to range from 600 to 700 per year.

Operating systems and processor manufactures aim to mitigate these kinds of attacks by realizing the $W \oplus X$ (Writable XOR Executable) security model, which prevents an adversary from executing malicious code by marking a memory page either writable or executable. Current Windows versions (such as Windows XP, Vista, or Windows 7) enable $W \oplus X$ (named data execution prevention (DEP) [43] in the Windows world) by default.

Return-oriented programming. However, *return-oriented programming (ROP)* [51] bypasses the $W \oplus X$ model because no code has to be injected and only code that resides in the process's image is executed. ROP is a generalization of *return-into-libc* attacks [54]. In a return-into-libc attack the adversary calls functions from the default UNIX C library libc without injecting malicious code. In contrast, ROP does not rely on functions available in libc, but instead uses small pieces of code within functions. Actually, the adversary calls no functions at all. For this purpose the adversary pushes onto the stack various return addresses, whereas each return address points to an instruction sequence in libc or in any other system library available in the process image. These instruction sequences are chained together to perform the adversary's attack.

The ROP attack method has been shown to be Turing-complete and its applicability has been demonstrated on a broad range of architectures: on PC platforms with Intel x86 [51] or RISC processors such as

*This is an updated and most recent version of the technical report. An earlier version was published before under the same report number in March 2010.

[1] National Institute of Standards and Technology

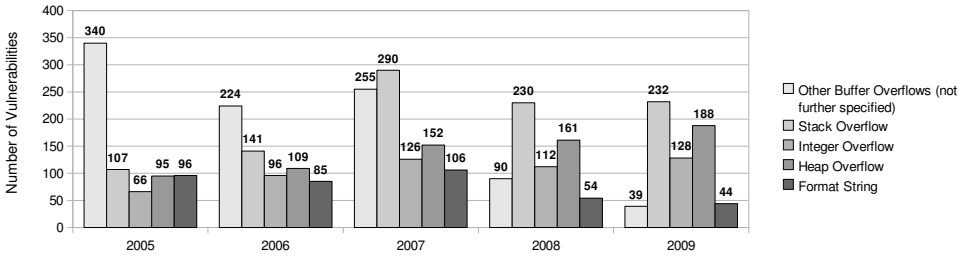


Figure 1: Buffer overflow vulnerabilities from 2005 to 2009

SPARC [8] and PowerPc [41]; on mobile devices with ARM based architectures [39]; and even on Harvard architectures such as AVR microcontrollers [24], or Z80 processors in voting machines [12]. Moreover, Hund et. al [33] presented a ROP based rootkit for the Windows operating system that bypasses kernel integrity protections and Bruschi et al. [49] showed how to bypass address space layout randomization (ASLR) [48] by using small ROP gadgets. Further, ROP-based attacks on well-established products such as Adobe Reader [37], Adobe Flashplayer [3], or Apple Quicktime Player [28] have been identified recently. Also recent attacks on Apple iPhone (that by default enables $W \oplus X$) are based on the principles of ROP, e.g., a recent iPhone jailbreak [30] and a SMS database attack [36].

Hence, we believe that ROP is a real threat to today's computing platforms. We expect malware designers to employ this technique in the near future when protection mechanisms such as $W \oplus X$ become widely deployed that would hinder conventional code injection attacks.

Although ROP is available on a broad range of architectures, it is particularly powerful on Intel x86 because of *unintended* instruction sequences. These *unintended* instruction sequences can be issued by jumping into the arbitrary position of a valid instruction resulting in a new instruction sequence. Such sequences can be found on Intel x86 due to variable-length instructions and unaligned memory access.

Existing Countermeasures. On the other hand, there exists a large number of proposals that aim to detect corruption of return addresses. These solutions can be categorized in *compiler*-based solutions [19, 57, 15, 40, 47]; *instrumentation*-based solutions such as securing function prologues and epilogues [16, 29], TRUSS (Transparent Runtime Shadow Stack) [53], Program Shepherding [38], Control-Flow Integrity (CFI) and XFI [1, 2]; and *hardware*-facilitated solutions [26, 25]. However, as we discuss in detail in related work (Section 6), the existing solutions suffer from the following shortcomings and practical deficiencies: They either cannot provide *complete detection* of ROP attacks [16, 29, 38] or require *side information* (e.g., debugging information [1, 2], or source code [19, 57, 15, 40, 47]) on the program's structure which are rarely provided in practice. Moreover, many of the instrumentation-based tools suffer from false positives because they do not handle exceptional cases such as C++ exceptions, Unix signals, or lazy binding. Finally, compiler-based solutions are from the end-user's perspective not always sufficient, because they will be only effective if *all* software vendors really deploy these compilers. However, in practice, software vendors often focus on performance rather than on security, and thus, many applications still suffer from various memory errors (see Figure 1) which allow adversaries to launch ROP attacks. In this paper, we aim to tackle the problems of existing solutions with the goal that end-users can immediately deploy a countermeasure against ROP.

Our contributions. In this paper, we present the design and implementation of *ROPdefender*, a practical tool that enforces return address protection and tackles the problem of existing solutions. We improve existing proposals by detecting unintended return instructions issued in a ROP attack without requiring any side information (e.g., source code or debugging). Our tool is built on top of the Pin framework [42], which provides just-in-time (jit) binary instrumentation. Pin is typically used for program analysis such as performance evaluation and profiling. Moreover, it has been used in [59] for a checksum-aware fuzzing tool and in [17] as dynamic taint analysis system. However, we use it for the purpose to detect ROP attacks. We therefore developed a new Pintool, our *ROPdefender*, that enforces return address checks at runtime. One of our main design goals was to create a practical tool that can be used immediately on mainstream platforms without the need to change hardware or the whole operating

system design. Hence, we aimed to adopt already existing techniques such as shadow stack [15, 57, 26] for return addresses, and the concept of binary instrumentation as used in taint tracking [46, 17] or return address protection [29, 16, 38]. Our contributions are in particular the following:

- **Defense without requiring side information:** *ROPdefender* requires no specific side information in order to enforce return address protection. It therefore does not suffer from practical constraints of CFI [1], which needs debugging information, or compiler-based approaches [19, 57, 15], which need the source code.
- **Flexibility and interoperability:** *ROPdefender* can be applied to complex multi-threaded applications such as Acrobat Reader or Mozilla Firefox. It can be deployed on Windows and Linux, and it requires in contrast to [25] no new hardware features.
- **Handling exceptions:** As we will discuss in Section 4, a sophisticated return address checker has to handle exceptions which break the calling convention. *ROPdefender* is able to handle Unix signals, C++ exceptions, non-local control transfers (i.e., setjmp/longjmp), and lazy binding used in Linux-based systems to avoid resolving function start addresses at runtime.
- **Security:** We are able to detect sophisticated attacks such as return-oriented programming (ROP) [51] which make use of unintended sequences (see Section 2.3 for an example). In contrast, similar rewriting based approaches [16, 29] only incorporate return address checks for *intended* returns.
- **Performance:** *ROPdefender* induces an overhead by a factor of about 2x. In Section 5.1 we discuss that comparable jit-based instrumentation tools add higher or comparable performance overhead.
- **Detection of real-world exploits:** As proof of concept we show in Section 5.2 that *ROPdefender* is able to detect a recent ROP-based exploit for Acrobat Reader [37] within 31 seconds. This exploit could not be detected by virus scanners until the signature of the exploit was known.

Our reference implementation of *ROPdefender* detects all ROP attacks based on returns. Further, it detects any attack that is based on corrupting a return address, e.g., conventional stack smashing [4] or return-into-libc [54]. However, recently Checkoway et al. [11] presented a new ROP attack which uses indirect jumps rather than returns. We stress that our current implementation of *ROPdefender* is currently not able to detect this new class of attack. We will discuss in Section 5.3 how such ROP attacks without returns can be addressed in the future. In our future work we aim to integrate *ROPdefender* into a control-flow integrity framework to also prevent ROP attacks without returns.

Outline. The remainder of this paper is organized as follows. Section 2 provides an overview to ROP attacks. We present the main idea of our approach and the architecture of *ROPdefender* in Section 3. We describe the details of our implementation in Section 4 and evaluate its performance and security in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Background on Return-Oriented Programming

An attack based on return-oriented programming (ROP) is usually introduced by means of a buffer overflow attack and uses principles of return-into-libc attacks. In the following we will present the basic idea of ROP and discuss the significance of unintended instruction sequences found on the x86 architecture.

2.1 Basic Attack Technique

The main goal of a conventional buffer overflow attack [4] is to subvert the usual execution flow of a program by redirecting it to a malicious code that was not originally placed by the programmer. Basically, the attack consists of two tasks: (i) injecting new malicious code in some writable memory area and (ii) changing a code pointer in such a way that it points to the injected malicious code. The preferred code pointer to run the attack is the return address on the stack.

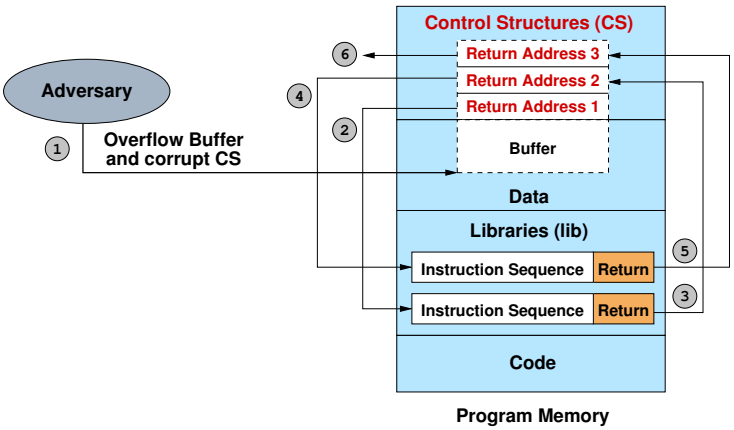


Figure 2: Simplified return-oriented programming attack

If the $W \oplus X$ model [48, 43] is enabled by the operating system (and supported by the hardware), the adversary will be no longer able to execute injected code, since a memory page is either marked writable or executable. Therefore, a more sophisticated attack was proposed using only pieces of code that resides in the process's image. The target for useful code pieces are especially within the Unix C library libc which is linked to nearly every Unix program and provides a number of useful functions (to the adversary). Hence, the return address points to a valid function in libc like *system* or *execve*. The attack is referred to as *return-into-libc* [54].

However, return-into-libc attacks are subject to some constraints. First, only those functions that reside in libc can be called by the adversary.[2] If the designers of libc would remove functions that are of particular interest to the adversary (e.g., *system*, *execve*, etc.) crafting a return-into-libc attack will become more difficult. Second, the adversary can only execute straight-line code, i.e., he/she can only invoke functions one after the other.

However, a more powerful class of attacks has been discovered recently [51], called return-oriented programming (ROP). ROP [51] can be seen as generalization of return-into-libc attacks that resolves the constraints of traditional return-into-libc attacks. ROP allows arbitrary computation without injecting new code and without calling any functions. ROP attacks are even applicable for established systems such as SPARC [8], Atmel AVR [24], voting machines [12], PowerPC [41] and even on ARM based architectures used in mobile devices [39].

In contrast to return-into-libc attacks, ROP attacks use small CPU instruction sequences instead of whole functions. These small instruction sequences range from two to five instructions and are chained together to perform a particular atomic task referred to as *gadget* like load, store or some arithmetic operation. Putting these gadgets together finally is referred to as return-oriented programming, and it can be used to build an attack that, for instance, launches a shell to the adversary as in a conventional buffer overflow attack.

Figure 2 illustrates the general ROP attack initiated by a buffer overflow. The difference to a conventional buffer overflow [4] is that the adversary does not need to inject its own code and is not restricted on functions available in libc but can use arbitrary instruction sequences of libc (or any other library or code segment that is linked to the address space of the process under attack) without calling functions explicitly. By chaining together the instructions sequences in a useful way, the adversary is able to perform arbitrary computation.

Figure 2 shows a simplified version of a program's memory layout consisting of a code section, libraries (lib), a data section and a control structure section (CS). In order to mount a ROP attack, the adversary exploits a memory-related vulnerability of a specific program. Traditionally, the ROP attack assumes a buffer overflow vulnerability on the stack. However, we show in Section 2.3 that a ROP attack can be also instantiated by other means. Hence, the adversary is able to overflow a local buffer and overwrite adjacent control-flow information of the CS section (the stack), e.g. the return address of the vulnerable

[2]Generally, it is also possible to use code from the text segment or any other shared library linked into the process image. However, still we have a defined set of code that can be used to craft a return-into-libc attack.

function (step 1). The adversary injects several return addresses each pointing to an instruction sequence in the lib section. Upon function return, execution is not redirected to the original calling function but instead to an instruction sequence in the lib section (step 2). This sequence is terminated by another return instruction which pops return address 2 from the CS section (step 3) and redirects execution to the next instruction sequence (step 4). This procedure is repeated until the adversary terminates the attack.

As shown above instruction sequences are chained together via return instructions. In general, the ROP attacks presented so far are all based on this principle, and hence, exploit return instructions or function epilogue sequences to redirect execution from one sequence to the next sequence [51, 8, 24, 39, 41, 12, 33, 49]. But note that recently Checkoway et al. [11] illustrated a ROP attack that is solely based on indirect jumps rather than returns. However, in this paper, we focus on conventional ROP attacks (based on return instructions), but we discuss in Section 5.3 how this new class of attacks can be addressed in the future.

2.2 Unintended Instruction Sequences

ROP attacks on the x86 architecture are particularly based on *unintended* instruction sequences. *Unintended* instruction sequences are not originally placed by the programmer (although formed and executed in a ROP attack). An unintended instruction sequence can be issued by jumping in the middle of a valid instruction resulting in a new instruction sequence never intended by the programmer. These sequences can be found in large amount on the x86 architecture because of the design principles of x86 as we will describe in the following. The Intel x86 or IA-32 architecture [34] is a well-established instruction set architecture deployed in personal computers. Shacham [51] outlines two outstanding properties of x86 that makes it particularly vulnerable to ROP attacks: (i) variable length instructions and (ii) unaligned memory access.

Consider for instance the following x86 code with the given intended instruction sequence, whereas the byte values are listed on the left side and the corresponding assembly code on the right side:

```
b8 13 00 00 00   mov $0x13,%eax
e9 c3 f8 ff ff   jmp 3aae9
```

If the interpretation of the byte stream starts two bytes later at byte 00 (at the third byte of instruction one), which is possible due to unaligned memory access, the following unintended instruction sequence would be executed by the processor:

```
00 00   add %al,(%eax)
00 e9   add %ch,%cl
c3      ret
```

In the intended instruction sequence the c3 byte is part of the second instruction. But if the interpretation starts two bytes later, the c3 byte will be interpreted as a return instruction. Shacham [51] especially makes use of unintended instruction sequence ending in a return instruction to enforce ROP attacks.

2.3 ROP Attacks Based on Unintended Instruction Sequences

There exists several compiler and instrumentation-based solutions that aim to detect corruption of return addresses. The main idea of these proposals is to keep copies of return addresses in a dedicated memory area, referred to as *shadow stack*. Upon function return, these solutions check if the return address has been modified. The idea was first proposed by Chiueh and Hsu [15] and was afterwards used in [26, 57, 16, 29, 53]. In the following we show that neither compiler nor instrumentation-based solutions securing returns only in function epilogues can prevent ROP attacks that are exclusively based on unintended instruction sequences.

The general attack steps for a ROP attack are depicted in Figure 3: (1) Taking control over the instruction pointer (IP) so that execution is redirected to the first instruction sequence and (2) let the the stack pointer (SP) point to the ROP payload (consisting of several return addresses and some data) to allow the chained execution of several instruction sequences. In order to avoid detection by countermeasures that secure returns in function epilogues, these two steps have to performed without using a return instruction in an intended function epilogue. Further, the instruction sequences executed

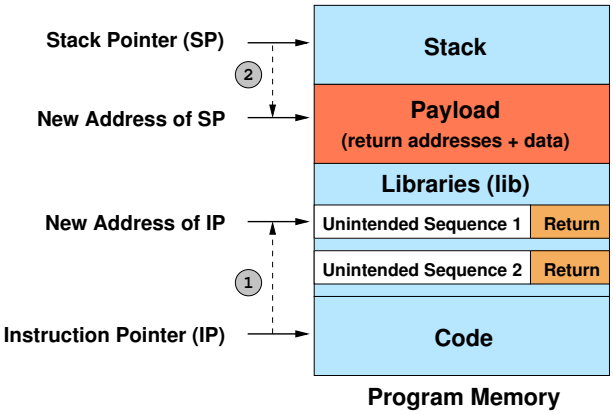


Figure 3: Initialization of a ROP attack: Taking control over the instruction pointer (IP) and the stack pointer (SP)

must all be unintended, i.e., have to end in an unintended return. That unintended instruction sequences are often found in x86 libraries has been shown in [51]. Indeed, the GALILEO algorithm proposed in [51] avoids intended function epilogues and [51] shows that a Turing-complete gadget set can be also derived from unintended sequences.

Usually, taking control over IP and SP can be performed through a conventional stack smashing attack [4] by overwriting a function's return address. However, this initial step, which is also required for ROP attacks, can be detected by return address checkers securing returns in function epilogues. On the other hand, stack smashing is not the only possibility to subvert the program flow. In the following we discuss several alternative attack techniques which allow the adversary to gain control over IP and SP without using intended returns.

Well-known vulnerabilities such as heap overflows [5], integer overflows [6] or format strings [27] allow an adversary to write arbitrary integer values into a program's memory space. Rather than overwriting a return address, the adversary could overwrite *pointers*, e.g., function pointers or entries of the Global Offset Table (GOT)[3]. If an adversary overwrites such a pointer, and the pointer is afterwards used as jump target (i.e., the value of IP is changed to the value stored in the corrupted pointer), execution will be redirected to code of the adversary's choice. Hence, such pointer manipulations allow an adversary to take control over IP. However, the adversary has also to ensure that SP points to the ROP payload in order to ensure the chained execution of the instruction sequences. In general, this can be performed by a stack-pivot sequence [20], which allows an adversary to change SP to an arbitrary value. For instance, this can be achieved in x86 by following sequences:

```
mov %eax,%esp; ret
xchg %esp,%eax; ret
mov %ecx, %esp; jmp *%edx
```

The first two sequences require the %eax register to contain the new value of SP (%esp). The first one moves %eax to %esp and the second one exchanges the contents of %eax and %esp. Since both sequences end in a return instruction (probably part of a function epilogue), they have to be unintended in order to avoid detection by countermeasures that secure returns in function epilogues. On the other hand the third sequence could be an intended instruction sequence, because it uses no return instruction and is probably not part of a function epilogue: It moves the content of %ecx to %esp and afterwards performs a jump to the address stored in %edx. Hence, %ecx must contain the new value of SP and %edx the address of the first sequence.

Another very simple technique to take control over IP and SP without corrupting a return address is through a setjmp vulnerability. For instance, this vulnerability has been exploited in [11] to instantiate a ROP attack without returns. Basically, setjmp and longjmp realize non-local control transfers. Setjmp stores besides some general-purpose registers, SP and IP, in a special data structure, called jmp_buf. Once longjmp is called, IP and SP are reset to the values stored in the jmp_buf structure. If the adversary is

[3]The GOT holds absolute virtual addresses to library functions.

able to overwrite the contents of jmp_buf before longjmp is called, e.g., by means of a buffer overflow, then he gets direct control over IP and SP without using a return instruction part of a function epilogue.

3 Our Approach to Detect/Prevent ROP

In this section we present the architecture we propose to defeat ROP attacks. We present our assumptions, the adversary model, and our approach and high-level architecture.

3.1 Assumptions

As we will elaborate in Section 6 on related work, the crucial issue about ROP attacks is that they can be constructed in such way to bypass countermeasures against buffer overflows or return-into-libc attacks. In the following we briefly discuss the main assumptions of our defense architecture and our adversary model.

1. **Access to side information:** We assume that we have *no access* to side information (e.g., source code or debugging information) while defeating ROP. These information are rarely provided in practice, impeding users to deploy defenses against ROP attacks.

2. **Platform Security:** We assume that the hardware and the operating system enforce the $W \oplus X$ security model. Thus, an adversary is forced to mount runtime attacks in a return-oriented way. This is reasonable, since today's processors feature a NX/XD (Non-Executable / Execution Disabled) Bit and many operating systems already enable enable $W \oplus X$ by default.

3. **Security of our tool:** The adversary cannot attack our tool itself or the underlying operating system kernel. If the adversary would be able to do so, any detection method could be circumvented or even disabled. Hence, we rely on other means of protection of the underlying trusted computing base, e.g., hardening of the operating system kernel, verification or extensive testing as well as load-time integrity checking of the software components belonging to our tool.

4. **Capabilities of the adversary:** The adversary is able to launch a ROP attack which *cannot* be detected by compiler-based solutions securing function epilogues. We described in Section 2.3 how such attacks can be constructed.

3.2 High-Level Description

Since we assume no access to source code (Assumption 2), we make use of a technique referred to as *instrumentation*. Basically, instrumentation allows us to add extra code to a program to observe and debug the program's behavior [44]. We use a *shadow stack* to store a copy of the return address (similar to, e.g., [15, 57, 26, 16, 29, 53]) once a function is called. We instrument all return instructions that are issued during program execution and perform a return address check. In contrast to existing shadow stack approaches, *ROPdefender* checks each return issued by the processor to detect even unintended instruction sequences, and it handles various special cases, which are not covered by existing solutions, but necessary for a practical defense tool.

According to the Intel x86 calling convention [34], return addresses have to be stored on the stack. A function call is performed through the `call` instruction, which automatically pushes the return address onto the top of the stack (TOS). After the called function has completed its task, it returns to the caller through a `ret` instruction, which pops the return address from the stack and redirects execution to the code pointed to by the return address. However, there are a few exceptions that violate the traditional calling convention and the function returns elsewhere. We discuss and categorize these exceptions in Section 4. For the moment, we assume that a function always returns to the address originally pushed by the call instruction. Nevertheless, our prototype implementation of *ROPdefender* also handles the exceptions as we detail in Section 4.

Our high-level solution for detecting ROP attacks is depicted in Figure 4. Before an instruction is executed by the processor, our solution intercepts the instruction and examines the instruction's type. First, we check if the current instruction is a call. If this is the case, we store a copy of the pushed return address in our shadow stack (transition 2a in Figure 4). Otherwise, if the instruction is a return

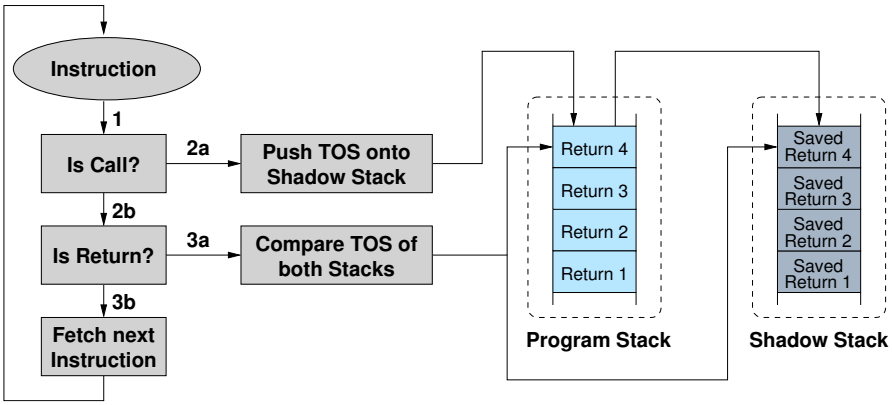


Figure 4: Our high-level approach

instruction, we check if the top return address on the shadow stack equals the return address on top of the program stack (transition 2b and 3a in Figure 4). If there is a mismatch, the return address has been corrupted or a calling exception occurred.

Our solution detects any return address violations: It does not only prevent ROP attacks. It also provides detection of all buffer overflow attacks which overwrite return addresses.

3.3 Tools and Techniques

As mentioned above, we use *instrumentation* to detect ROP attacks. Generally, instrumentation can be performed at runtime, at compile-time, or within the source code. For our purpose we focus on dynamic binary instrumentation at runtime to avoid access to side information, e.g., source code, debugging information, etc. Generally, there are two classes of dynamic binary instrumentation frameworks: (i) probe-based and (ii) those using a just-in-time compiler (jit-based).

Probe-based instrumentation used in DynInst [9], Vulcan [23] or DTrace [10] enforces instrumentation by replacing instructions with the so-called trampoline instructions in order to branch to instrumentation code. DTrace, for instance, replaces instrumented instructions with special trap instructions that once issued generate an interrupt. Afterwards the instrumentation code is executed.

Jit-based instrumentation frameworks like Valgrind [45], DynamoRIO [7], and Pin [42] use a just-in-time compiler. In contrast to probe-based instrumentation no instructions in the executable are replaced. Before an instruction is executed by the processor, the instrumentation framework intercepts the instruction and generates new code that enforces instrumentation and assures that the instrumentation framework regains control after the instruction has been executed by the processor.

We use jit-based instrumentation since it allows us to detect sophisticated ROP attacks based on unintended instruction sequences (see Section 2.3 for an example): It allows to intercept each instruction before it is executed by the processor, whether the instruction was intended by the programmer or not. In contrast, probe-based instrumentation frameworks rewrite instructions ahead of time with trampoline instructions and consequently instrumentation is only performed if the trampoline instruction is really reached.

3.4 General Architecture

We incorporate *ROPdefender* directly into the dynamic binary instrumentation (DBI) framework. The DBI framework as well as the operating system are part of our trusted computing base (TCB). Hence, we assume that an adversary cannot attack the *ROPdefender* software itself or the underlying operating system kernel (see Assumption 4). Figure 5 depicts our proposed architecture to effectively defeat ROP attacks.

The general workflow is as follows: The program is loaded and started under the control of the DBI framework. The DBI framework ensures that (i) each instruction of the program is executed under control of the DBI and (ii) all instructions are executed according to the *ROPdefender* specific instrumentation

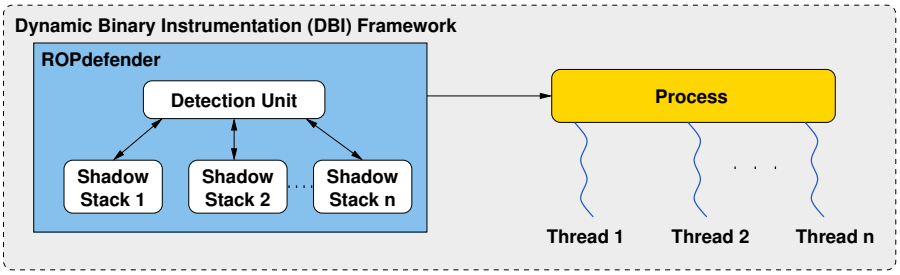


Figure 5: General architecture of *ROPdefender*

code. More precisely, the current instruction of the process is intercepted by the DBI framework and forwarded to *ROPdefender* which then enforces the return address check.

ROPdefender consists of a detection unit and several shadow stacks. The detection unit pushes/pops return addresses onto/from the connected shadow stacks. Further, the detection unit is responsible for enforcing the return address check. The reason why *ROPdefender* maintains multiple shadow stacks is that a process may launch several execution threads. If all threads would share one shadow stack, false positives would arise, since the threads would concurrently access the shadow stack.

4 Implementation

In this section we describe implementation details of our framework and our *ROPdefender*. For our implementation we used the jit-based binary instrumentation framework Pin (version 2.8-33586) and the Linux Ubuntu OS (version 10.04). We also implemented our tool on Windows XP, but we describe our implementation details and exception handling in the following for the Linux Ubuntu OS. Further, our implementation of the *ROPdefender* detection unit is one C++ file consisting of 165 lines of code.

The rationale behind using Pin [42] was that in [42] Cohn et al. benchmarked well-known jit-based DBI frameworks and concluded that Pin achieves the best performance among them. Pin [42] is typically used for program analysis such as performance evaluation and profiling.[4] Intel uses Pin in the Intel Parallel Studio [35] for memory and thread checking or bottleneck determination. However, we use this binary instrumentation framework for the purpose of detecting ROP attacks.

4.1 Binary Instrumentation Architecture

Figure 6 shows the instantiation of our architecture.

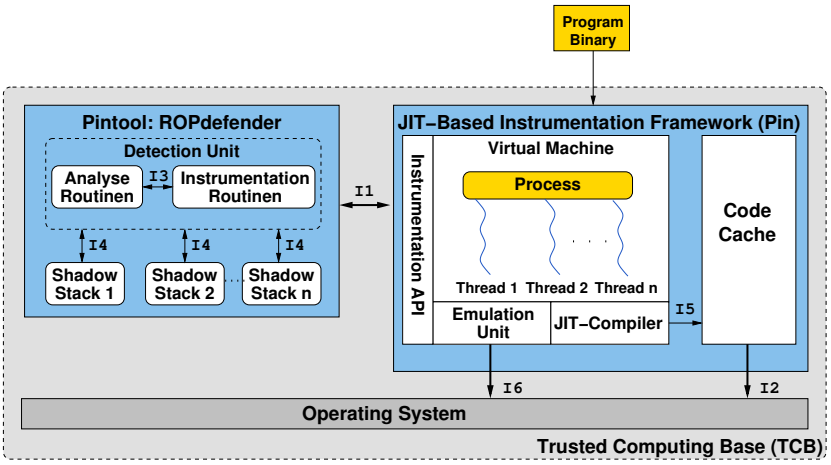


Figure 6: Implementation of *ROPdefender* within the binary instrumentation framework Pin

[4]Note that Pin has been also used in [59] for a checksum-aware fuzzing tool and in [17] as dynamic taint analysis system.

Pin. Pin itself has mainly two components: (i) a code cache and (ii) a Virtual Machine (VM) which contains a JIT compiler and an emulation unit. A program instrumented by Pin is loaded into the VM. The JIT compiler enforces instrumentation on the program at runtime. The resulting instrumented code is stored in the code cache in order to reduce performance overhead if code pieces are invoked multiple times.

Pin is configured via Pintools. Basically, Pintools allow us to specify our own instrumentation code. The JIT compiles instructions according to the Pintool. Pintools can be written in the C/C++ programming language. Effectively, here is the place where we implement our *ROPdefender*. After Pin is loaded and initialized, it initializes the *ROPdefender* detection unit. Then the program which we want to protect is started under the control of Pin. When a program is started, Pin intercepts the first *trace* of instructions and the JIT compiles it into new instructions in order to incorporate instrumentation code. A trace is a sequence of instructions terminated by an unconditional branch. Trace instrumentation allows to instrument an executable one trace at a time. The trace consists of several basic blocks, whereas each block is a single entry and a single exit (any branch) sequence of instructions. Instrumenting blocks is more efficient than instrumenting each instruction individually. Afterwards, the compiled code is transferred to a code cache over the interface I5 that finally forwards the compiled instructions to the operating system through interface I2. If a sequence of instructions is repeated, no recompilation is necessary and the compiled code can directly be taken from the code cache. The emulation unit is necessary for those instructions that cannot be executed directly (e.g., system calls). Such instructions are forwarded to the operating system over interface I6.

Instrumentation and Analysis Routines. According to Figure 4 in Section 3.2, we specified two instrumentation routines that check if the current instruction is a call or a return instruction. Further, we defined two analysis routines that perform the actions and checks according to the steps 2a and 3a in Figure 4. To implement a shadow stack for each thread we additionally use the C++ stack template container. To avoid that one thread accesses the shadow stack of another thread, we use the thread local storage (TLS) from the Pin API, whereas each thread must provide a key (created at thread creation) to access its TLS. Elements can be pushed onto and popped off the shadow stack as for the usual stack in program memory. The instrumentation routines of our *ROPdefender* use the inspection routines *Ins_IsCall(INS ins)* and *Ins_IsRet(INS ins)* provided by the Pin API to determine if the tail instruction of the current basic block is a call or a return instruction. If the instruction is a call instruction, then we invoke an analysis routine (step 2a) that pushes the return address onto the appropriate shadow stack. Otherwise, if the instruction is a return instruction, then a second analysis routine checks if the return address the program wants to use equals to the address at the top of the corresponding shadow stack (step 3a).

4.2 Handling Exceptions

As mentioned in Section 3.2, the common calling convention assumes that an invoked function will always return to the address pushed onto the stack by the calling function. However, our experiments have shown that there are a few exceptions violating this calling convention. These exceptions can be categorized into three classes: (Class 1) A called function does not return, i.e., the control is transferred out of the function before its return instruction has been reached. (Class 2) A function is invoked without explicitly using a call instruction. (Class 3) A different return address is computed while the function is running.

Due to all these exceptions, developing an efficient and also practical return address protection tool is not straightforward. Although many proposals address the first class of exceptions (e.g., [15, 16, 29, 53, 38]), there exists no proposal addressing Class 2 and 3. In contrast, our *ROPdefender* handles all above mentioned classes of exceptions. Note that the exceptions described below are the most well-known ones (for instance, *ROPdefender* does not raise any false positive for a whole SPEC CPU benchmark run), and there may be further exceptions in practice which may raise false positives. However, we believe that additional exception handling can be easily integrated into *ROPdefender* based on the techniques discussed below.

Class 1: Setjmp/Longjmp. For the case that the instrumented program uses the system calls *setjmp* and *longjmp* then we expect false positives, because these functions allow to bypass multiple stack frames

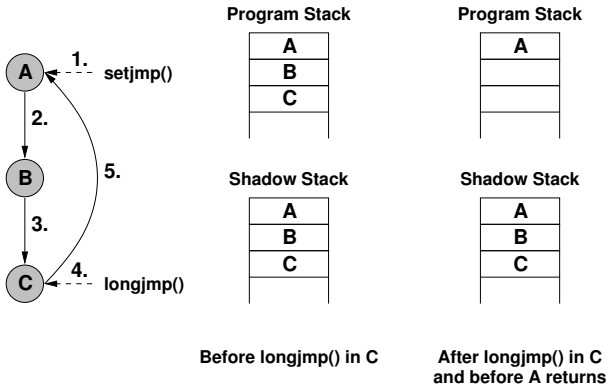


Figure 7: Example for setjmp and longjmp

and violate therefore the usual calling convention with a non-local control transfer that can be seen as a non-local goto. Figure 7 depicts a situation where a non-local control transfer occurs. Actually, it depicts a function calling sequence. Function A starts execution and issues the *setjmp* system call which saves the current stack state. Afterwards, A calls B and B calls C. Within the execution of C, the *longjmp* system call is issued that restores the stack state to the state as it was at the time the *setjmp* system call was invoked. Thus, C does not return to B, instead a non-local goto to A is enforced and the program stack will no longer contain the stack frames of B and C.

Upon return of A, the return address on the program stack does not match the return address on the shadow stack, because the stack frames of B and C are only removed from the program stack but not from the shadow stack, which results in a false positive. To avoid a false positive, *ROPdefender* uses a strategy similar to RAD [15] popping continuously return addresses off the shadow stack until a match is found or until the shadow stack is empty. The latter case would indicate a ROP attack.

Class 2: Unix signals and lazy binding. A typical example for the second class are Unix signals. Generally, signals are used in Unix-based systems to notify a process that a particular event (e.g., segmentation fault, arithmetic exception, illegal instruction, etc.) have occurred. Once a signal has been received, the program invokes a signal handler. If such a signal handler is implemented through the *signal* function, then execution is redirected to the handler function without a `call` instruction. Hence, if the signal handler returns, *ROPdefender* would raise a false positive, because the return address of the handler function has not been pushed onto the shadow stack. However, the relevant return address is on top of the program stack before the signal handler is executed. To avoid a false positive, we use a signal detector (provided by the Pin API) in order to copy the return address from the program stack onto our shadow stack when a signal is received.

Another typical example for Class 2 is lazy binding which uses a return instruction to enforce a jump to a called function. Lazy binding is enabled by default on UNIX-based systems. It decreases the load-time of an application by delaying the resolving of function start addresses until they are invoked for the first time. Otherwise, the dynamic linker has to resolve all functions at load-time, although they may be never called. On our tested Ubuntu system, lazy binding involves the functions *_dl_rtld_di_serinfo* and *_dl_make_stackexecutable*, which are both part of the dynamic linker library *linux-ld.so*. After *_dl_rtld_di_serinfo* resolves the function's address, it transfers control to the code of *_dl_make_stackexecutable* by a jump instruction. Note that *_dl_make_stackexecutable* is not explicitly called. However, *_dl_make_stackexecutable* redirects execution to the resolved function through a return instruction (rather than through a jump/call). To avoid a false positive, we push the resolved function address onto our shadow stack before the return of *_dl_make_stackexecutable* occurs. Our experiments have shown that the resolved address is stored into the %eax register after *_dl_rtld_di_serinfo* returns. Hence, we let *ROPdefender* push the %eax register onto our shadow stack when *_dl_rtld_di_serinfo* returns legally.

Class 3: C++ Exceptions. Another type of exceptions are those where the return address is computed while the function executes, whereas the computed return address completely differs from the

return address pushed by the `call` instruction. A typical example for this are GNU C++ exceptions[5] with stack unwinding. Basically, C++ exceptions are used in C++ applications to catch runtime errors (e.g., division by zero) and other exceptions (e.g., file not found). A false positive would arise if the exception occurs in a function that cannot handle the exception. In such case, the affected function forwards the exception to its calling function. This procedure is repeated until a function is found which is able to handle the exception. Otherwise the default exception handler is called. The invoked exception handler is responsible for calling appropriate destructors[6] for all created objects. This process is referred to as stack unwinding and is mainly performed through the GNU unwind functions *_Unwind_Resume* and *_Unwind_RaiseException*. These functions make a call to *_Unwind_RaiseException_Phase2* that computes the return address and loads it at memory position `-0xc8(%ebp)`, i.e., the %ebp register minus 200 (0xc8) Bytes points to the return address. In order to push the computed return address onto our shadow stack, *ROPdefender* copies the return address at `-0xc8(%ebp)` on our shadow stack after *_Unwind_ RaiseException_Phase2* returns legally.

5 Evaluation

In this section we evaluate the performance of *ROPdefender*, show how *ROPdefender* detects recent ROP-based exploits, and finally, we discuss ROP attacks exploiting indirect jumps.

5.1 Performance

To evaluate the overall performance, we have measured the CPU time of *ROPdefender*. We compare our results to normal program execution and to execution with Pin but without instrumentation. Our testing environment was a 3.0 GHz Intel Core2 Duo E6850 machine running Ubuntu 10.04 (i386) with Linux kernel 2.6.28-11 and Pin version 2.8-33586. We ran the integer and floating-point benchmarks from the SPEC CPU2006 Suite [56] using the reference inputs. Figure 8(b) and 8(a) depict our benchmark results.

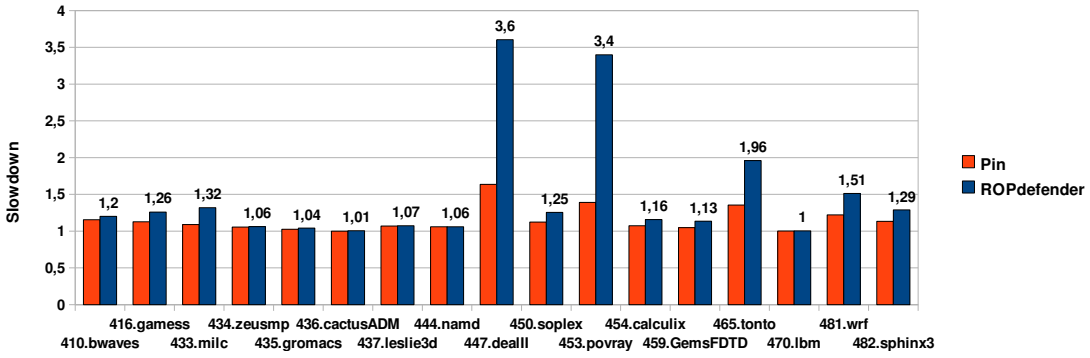
Pin without instrumentation. The Pin framework itself induces an average slowdown of 1.58x for integer computations and of 1.15x for floating point computations. The slowdown for integer computations ranges from 1.01x to 2.35x. In contrast, for floating point computations the slowdown ranges from 1.00x to 1.64x.

Pin with *ROPdefender*. Applications under protection of our *ROPdefender* run on average 2.17x for integer and 1.49x for floating point computations slower than applications running without Pin. The slowdown for the integer benchmarks ranges from 1.01x to 3.54x, and for the floating point from 1.00x to 3.60x. *ROPdefender* adds a performance overhead of 1.49x for integer and 1.24x for floating point computations in average compared to applications running under Pin but without instrumentation. We compared *ROPdefender* with other known tools such as the dynamic taint analysis systems DYTAN [17] (also based on Pin) or TaintCheck [46] (based on Valgrind). According to the results in [17, 46], applications running under these tools are from 30x to 50x times slower which is enormously higher compared to *ROPdefender*. Also DROP [14] causes an average slowdown of 5.3x.

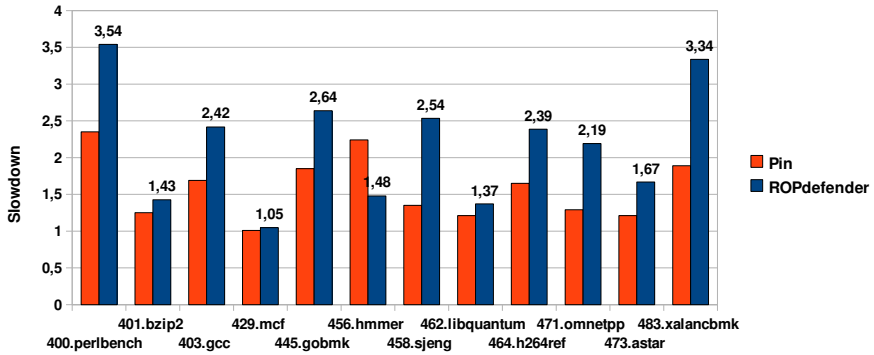
To increase the performance of *ROPdefender*, we can either improve the Pin framework itself or optimize the *ROPdefender* detection unit. The Pin developers are mainly concerned to optimize their framework in order to achieve better performance. Hence, we believe that performance of Pin will be continuously improved. Our detection unit avoids to check whether each instruction issued is a call/return by using trace instrumentation (see Section 4). Hence, we only check if the tail instruction of the current basic block is a call or return.

[5] Although we focus on the implementation of C++ exceptions with the GNU compiler, we believe that our solution can be also adopted to operating systems using a different compiler.

[6] Destructors free the memory and resources for class objects and members.



(a) Floating Point Benchmarks



(b) Integer Benchmarks

Figure 8: SPEC CPU2006 Benchmark Results

5.2 Case Study

ROPdefender is able to detect and prevent available real-world ROP exploits. As a use-case, we apply it to a recent Adobe Reader exploit [37]. Generally, the attack in [37] exploits an integer overflow in the libtiff library, which is used for rendering TIFF images. The attack works as follows: By means of ROP it allocates new memory marked as writable *and* executable in order to bypass $W \oplus X$. Afterwards, the *memcpy* function is called to copy malicious code (stored in the PDF file itself) into the new memory area. Finally, execution is redirected to the malicious code, which could, for instance, launch a remote shell to the adversary. The exploit could not be recognized by virus scanners because its signature was not yet available. Since *ROPdefender* does not rely on such side information, it can immediately detect the attack.

In practice, an adversary will send the malicious PDF file to the victim user via an e-mail. The user opens the PDF file and thus, a remote shell is launched to the adversary. In order to apply *ROPdefender*, we adapted it to Windows. Instead of opening the file directly, we opened the file under the control of *ROPdefender*. Since the attack triggers an integer overflow and afterwards uses ROP instruction sequences (ending in returns), *ROPdefender* can successfully detect the attack at the moment the first sequence issues a return. Afterwards *ROPdefender* immediately terminates the application and informs the user.

In total, it takes 31 seconds until *ROPdefender* detects the attack. Table 1 shows a snapshot of *ROPdefender*'s output when it is applied to the exploit. The function from where the return instruction originated and the value of the instruction pointer (%eip) are shown in column 1 and 2. Sometimes Pin is not able to identify the precise function name. In such case, the default function name `.text` is assigned. The expected return address (placed on top of the shadow stack) and the malicious return address (used by the adversary) are shown in column 3 and 4.

As can be seen in the first row, the first return address mismatch occurs at address `0x070072F7`. The expected return address at that time is `0x7C921E29`. However, Adobe Reader aims to return to address

Function Name	Instruction Pointer	Expected Return	Malicious Return
.text	0x070072F7	0x7C921E29	0x20CB5955
unnamedImageEntryPoint	0x070015BB	NULL	0x070072F8
.text	0x0700154D	NULL	0x070015BC
.text	0x070015BB	NULL	0x0700154F
.text	0x07007FB2	NULL	0x070015BC
unnamedImageEntryPoint	0x070072F7	NULL	0x07007FB4
.text	0x070015BB	NULL	0x070015BC
BIBLockSmithAssertNoLocksImpl	0x0700A8AC	NULL	0x0700A8B0
.text	0x070015BB	NULL	0x070015BC
BIBLockSmithAssertNoLocksImpl	0x070072F7	NULL	0x0700A8B0
unnamedImageEntryPoint	0x070052E2	NULL	0x070072F8
BIBInitialize4	0x07005C54	NULL	0x070052E4
...	...	...	...

Table 1: Detection of ROP Attack on Acrobat Reader with *ROPdefender*

`0x20CB5955`. *ROPdefender* now has to check if either a return address attack or a setjmp/longjmp exception (see Section 4) occurred. Hence, *ROPdefender* pops continuously return addresses from the shadow stack until a match is found. Since the malicious return address `0x20CB5955` is not part of our shadow stack, *ROPdefender* will report the return address attack as shown in the first row of Table 1. To show that *ROPdefender* detects all malicious returns issued in the exploit, we temporarily allow the exploit to continue. As can be seen from Table 1, *ROPdefender* also detects the following malicious returns. All following expected return addresses are `NULL`, because the shadow stack is empty after the first mismatch.

5.3 Discussion and Future Work

Recently, Checkoway et al. [11] presented a new ROP attack for Intel x86 [13] and ARM architectures [21] that is only based on indirect jump instructions rather on returns. On Intel x86, each instruction sequence ends with an indirect jump to a pop-jump sequence which simulates the return and acts as a trampoline after each instruction sequence, e.g., `pop %edx; jmp *(edx)`: It pops the top of the stack into %edx and afterwards jumps to the popped address. However, such sequences (even unintended ones) rarely occur in practice. Therefore, Checkoway et al. [13] introduced the *bring your own pop-jump (BYOPJ)* paradigm, which assumes the availability of a pop-jump sequence in the target program or in one of its libraries. In order to defend against the new ROP attack, *ROPdefender* has to decide at runtime if a jump target is a legal one or not. Since there exists no convention regarding the target of an indirect jump instruction (in contrast to returns), it seems impossible to defend against such ROP attacks without having some information about the program's structure. However, ROP attacks based on indirect jumps share some characteristics that are unique and rarely found in ordinary programs. As mentioned above and shown in [13], pop-jump sequences are uncommon in ordinary programs, but the ROP attacks without returns invoke such a sequence after each instruction sequence. Hence, it might be possible to extend *ROPdefender* with a frequency measurement unit (as proposed in [22] and [14]). Actually, we measured the frequency of indirect jumps with the SPEC CPU Benchmark suite. Approximately each 153th instruction is on average an indirect jump. Return instructions occur more frequently than indirect jumps, in fact, each 59th instruction is a return instruction.[7] The concrete implementation presented in [13] make even use of two indirect jumps within three instructions (`jmp *x; pop *y; jmp *y`). Thus, frequency analysis against ROP attacks based on indirect jumps can be deployed as first ad-hoc solution. However, if the adversary issues a longer instruction sequence in between he might be able to bypass such a defense. Moreover, the adversary might be also able to use other return-like instructions such as indirect calls and thus bypass a solution that looks only for returns and indirect jumps. It remains open if an effective countermeasure against ROP without returns can be deployed without knowing the structure of the target program. However, *ROPdefender* with an extended frequency measurement unit rules out already many ROP attacks even under the assumption that an adversary successfully subverted the control flow by other means and afterwards only invokes unintended instruction sequences.

[7] Note that the results refer to C/C++ and FORTRAN compiled code. Other languages such as Forth might use indirect jumps more frequently.

6 Related Work

We explore well-established countermeasures against buffer overflow attacks and discuss to what extend they can be used in order to defeat ROP attacks.

6.1 Type Safe Languages

Type-safe languages like Java or C# are not as vulnerable to buffer overflow attacks as software written in the C/C++ language. Type safety means that the operations performed on a variable are only those as described by the type of the variable. However, in today's operating systems we still have a large amount of software that is written in unsafe languages.

6.2 The $W \oplus X$ Model

This protection scheme prevents conventional buffer overflow attacks that redirect execution to injected code by marking a memory page either executable or writable. $W \oplus X$ can be enabled for Unix-based operating systems by a kernel patch provided by PaX [48] and is enabled by default on recent Windows operating systems such as Windows Vista and Windows Seven [43]. Even mainstream semiconductor chip makers like AMD and Intel provide recently a new bit referred to as Non-Executable Bit (NX/XD) that can be enabled on each memory page. $W \oplus X$ cannot prevent ROP attacks that use code residing in the process's image (like libc) that is marked executable.

6.3 Randomization

Address Space Layout Randomization (ASLR) [48, 32] aims to prevent return-into-libc attacks by randomizing base addresses of code segments. Since the adversary has to know the precise addresses of all instruction sequences, this approach seems to effectively prevent ROP attacks. However, it has been shown that ASLR can be bypassed by using derandomization attacks [52] or through information leakage attacks targeting in particular web browsers such as Mozilla Firefox or Internet Explorer [55, 50, 58]. Moreover, some libraries or parts of the code segment may not be ASLR-compatible allowing adversaries to find enough useful instruction sequences to launch a ROP attack [49]. The latter uses ROP gadgets that are placed at fixed locations in the code segment and launches a return-into-libc attack. This is possible because the absolute base address of a library can be computed from the data stored in the Global Offset Table (GOT). To prevent this attack, the authors of [49] propose to encrypt the function addresses in the GOT at runtime. However, their solution does not support lazy binding and cannot detect return address attacks beyond exploiting the GOT. In contrast, *ROPdefender* can detect all ROP-based attacks even adversaries are able to bypass ASLR.

6.4 Compiler Extensions

Various compiler extensions were proposed to mitigate return address attacks. StackGuard [19] places a dummy value, referred to as *canary*, below the return address on the stack. Before a function returns, a check is enforced that proves whether the canary value has been overwritten or not. ProPolice [31] reorders local variables to place buffers below the saved base pointer and places a guard value between the buffers and the saved base pointer. Thus, if a buffer overflow occurs the local variables and pointers will not be overwritten. Only the return address and the saved base pointer are overwritten which are still protected by a canary. A more general approach, called PointGuard [18], encrypts all pointers and only decrypts them when they are loaded into CPU registers. Hence, the adversary has only access to encrypted pointers stored on memory. Close to our approach, Stack Shield [57] and Return Address Defender (RAD) [15] guard the return addresses by holding copies of them in a safe memory area.

However, compiler-based solutions require recompilation and access to the source code. In contrast *ROPdefender* requires no access to source code and therefore allows end-users to immediately deploy a countermeasure against ROP. Further, none of them is able to detect ROP attacks based on unintended instruction sequences (see Section 2.3 for an example). Nevertheless, in our approach we use the idea of keeping a copy of the return address onto a shadow stack as used in [57, 15].

Finally, two compiler-based solutions were developed in parallel to our work that provide specifically protection against ROP attacks [40, 47]. Li et al. [40] developed a compiler-based solution against kernel rootkits that are based on the principles of ROP [33]. First, it eliminates all unintended return instructions through code transformation. Second, the intended return instructions are protected by a technique referred to as return indirection: The call instructions push a return index onto the stack which points to a return address table entry whereas the return address table contains valid return addresses the kernel is allowed to use. Hence, the adversary can only use return addresses which are included in the return address table. The solution in [40] is complementary to our work, because it provides protection at the kernel-level, whereas *ROPdefender* targets ROP attacks on the application-level. However, in contrast to their work *ROPdefender* requires no access to source code and also addresses exceptional cases which might occur during ordinary program execution.

A recent and noteworthy compiler-based approach is G-Free [47] that defeats ROP attacks through gadget-less binaries. In contrast to the aforementioned approach and to *ROPdefender*, G-Free is also able to detect ROP attacks that are based on indirect jumps. Basically, G-Free eliminates all unintended instruction sequences by inserting an alignment sled (i.e., a byte stream consisting of nop instructions) before all instructions which include byte values that might be useful for an unintended return or indirect jump/call. Intended return instructions are protected by encrypting return addresses against a random cookie created at runtime. Finally, upon function entry, a function-unique cookie is pushed onto the stack for those functions that contain (intended) indirect jump or call instructions. Once an indirect jump/call occurs, the cookie will be decrypted and only if the decryption is successful, the branch will be allowed. Hence, this approach prevents the adversary from executing indirect jumps/calls in functions that were not explicitly called before. However, although *ROPdefender* does not yet provide protection against ROP without returns, it can be immediately deployed by end-users, because it does not require access to side information.

6.5 Instrumentation-Based Solutions

Securing function epilogues. There are approaches [29, 16] that aim to detect malicious changes of return addresses by using instrumentation techniques without requiring source code. Both approaches rewrite function prologue and epilogue instructions to incorporate a return address check on each function return. Chiueh et al. [16] use static instrumentation, i.e., the binary is disassembled and rewritten before it is executed. However, accurate disassembly ahead of time is error-prone and difficult to achieve (as analyzed in [16]). Gupta et al. [29] use probe-based instrumentation instead. However, as we already described in Section 2.3, both approaches are not able to detect ROP attacks that use unintended instruction sequences, because they only instrument intended function epilogues.

Control Flow Integrity. Control Flow Integrity (CFI) [1] used in XFI [2] guarantees that program execution follows a Control Flow Graph (CFG) created at load-time. XFI requires modification, i.e., rewriting of the binary in order to add the so-called individual label instructions that indicate potential and legal branch targets.[8] During program execution, any branch instruction has to be instrumented in order to check if the destination of the branch is pointing to a valid label instruction. Moreover, if a function returns to its caller, the stack pointer has to point to a valid return address.

Regarding return address protection, the XFI's rewriting engine first disassembles the binary in order to find all return instructions. Afterwards it rewrites the returns to embed additional instrumentation code that enforces a runtime check on the return address. Therefore XFI only instruments intended return instructions and if an adversary is able to launch the first instruction sequence in a ROP attack that ends in an unintended return instruction (which might be impossible if XFI guarantees that each branch instruction is instrumented correctly), XFI will not be able to check if the return address at this moment points to a valid label instruction. Besides, the binary instrumentation framework Vulcan [23] used by XFI is not publicly available and is restricted to the Windows operating system. Moreover, to build the CFG, XFI requires information on the program's structure which are extracted from Windows debugging information files (PDB files). However, such PDB files are not provided by default for each application. Contrary, *ROPdefender* needs no information to enforce detection of ROP attacks and is based on the open source Pin framework.

[8] They are similar to a nop instruction. Abadi et al. [1] propose the prefetchnta instruction.

Measuring frequency of Returns. Chen et al. [14] and Davi et al. [22] exploit jit-based instrumentation to detect ROP attacks. Both solutions count instructions issued between two return instructions. If short instruction sequences are issued three times in a row, they report a ROP attack. To bypass such a defense, an adversary could enlarge the instruction sequences or enforce a longer instruction sequence after, for instance, each second instruction sequence.

Just-in-Time Instrumentation. Program Shepherding [38] is based on the jit-based instrumentation framework DynamoRIO and monitors control flow transfers to enforce a security policy. In contrast to *ROPdefender*, it also instruments jump and call instructions from one code segment to another, e.g., a jump from application code to a shared library should be only allowed if it targets a valid entry point in the shared library. Moreover, as part of its restricted control-flow policy it provides the following return address protection: It guarantees that a return only targets an instruction that is preceded by a call instruction. Hence, the adversary can only invoke instruction sequences where the first instruction is preceded by a call instruction. Although this prevents basic ROP attacks, it is still possible to construct ROP attacks and to manipulate return addresses, because Program Shepherding does not ensure that a return really targets its original destination (e.g., the calling function). Since each library linked into the program's memory space contains various call instructions, the adversary still can return and invoke various instruction sequences without being detected by Program Shepherding. In contrast, *ROPdefender* detects any return address manipulation and therefore completely prevents the conventional ROP attacks that are based on returns. Moreover, Program Shepherding only handles the special case of setjmp/longjmp, whereas *ROPdefender* also handles exceptions of Class 2 and 3 (see Section 4). Another tool based on DynamoRIO is TRUSS (Transparent Runtime Shadow Stack) [53]. Similar to our approach, return addresses are pushed onto a shadow stack and a return address check is enforced upon a function return. Due to jit-based instrumentation, TRUSS is also able to detect unintended sequences issued in a ROP attack. However, the DynamoRIO framework does not allow to instrument a program from its very first instruction. It depends on the `LD_PRELOAD` variable which is responsible for mapping the DynamoRIO code into the address space of the application. Further, similar to Program Shepherding TRUSS does not handle exceptions of Class 2 and 3.

Taint Tracking. Dynamic taint analysis based on jit-based instrumentation (e.g., [46, 17]) marks any untrusted data (e.g., user input) as tainted. Tainted data could be user input or any input from an untrusted device or resource. After marking data as tainted, taint analysis tracks the propagation of tainted data, and alerts or terminates the program if tainted data is misused. Misuse of the tainted data is, for instance, using the tainted data as jump/call or return target. This mechanism induces a high performance overhead (30x to 50x for TaintCheck [46] and DYTAN [17]). However, we believe that *ROPdefender* can also be incorporated into existing taint analysis systems.

6.6 Hardware-Facilitated Solutions

In [25] an embedded microprocessor is adapted to include memory access control for the stack, which is split into data-only and call/return addresses-only parts. The processor enforces access control that does not allow to overwrite the call/return stack with arbitrary data. This effectively prevents ROP attacks. However, the approach is only demonstrated on a modified microprocessor and cannot be transferred easily to complex instruction CPUs like Intel/AMD architectures. Moreover, we do not expect CPU-integrated protection against ROP to appear in the near future. In contrast, our solution is software-based and works with general purpose CPUs and operating systems. Another hardware-facilitated solution available on SPARC systems is StackGhost [26]. StackGhost is based on stack cookies that are XORed with return addresses at function entry and XORed again upon function return. The design of StackGhost also includes a return address stack (similar to our shadow stack), but to the best of our knowledge, this has not been implemented and benchmarked. Further, StackGhost depends on specific features, which are unique to SPARC and which, according to [26], cannot be easily adopted to other hardware platforms.

7 Conclusion and Future Work

Return-oriented programming (ROP) as presented by Shacham is a powerful attack that bypasses current security mechanisms widely used in today's computing platforms. The ROP adversary is able to perform Turing-complete computation without injecting any new code. Further, he is able to execute instruction sequences that were never intentionally placed by the programmer.

The main contribution of our work is to present an effective and practical countermeasure against the conventional ROP attack without requiring access to side information. In this paper, we presented our *ROPdefender* that fulfills accurately these requirements and that is able to detect/prevent even ROP attacks that are based on *unintended* instruction sequences. For this, we exploited the idea of duplicating return addresses onto a shadow stack and the concept of jit-based binary instrumentation to evaluate each return instruction during program execution. In addition, we showed how to handle various exceptional cases that can occur during program execution in practice.

ROPdefender induces a performance overhead by a factor of 2x which cannot be expected by time-critical applications. Moreover, we need protection against return address attacks targeting the operating system that *ROPdefender* relies on. But, *ROPdefender* is already a practical solution that can be immediately deployed by end-users to protect applications against ROP attacks (based on return instructions) without requiring access to source code or any other side information. Currently, we are working on a countermeasure against ROP attacks without returns and on a countermeasure against ROP for embedded systems based on the ARM architecture.

Acknowledgements

We thank Hovav Shacham and Stephen Checkoway for the fruitful discussions on return-oriented programming attacks based on indirect jumps.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[2] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, George C. Necula, and Michael Vrable. XFI: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[3] Adobe Systems. Security Advisory for Flash Player, Adobe Reader and Acrobat: CVE-2010-1297. http://www.adobe.com/support/security/advisories/apsa10-01.html, 2010.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

[5] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.

[6] blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002.

[7] Derek L. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. http://groups.csail.mit.edu/cag/rio/derek-phd-thesis.pdf, 2004. PhD thesis, M.I.T.

[8] 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. ACM, 2008.

[9] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[10] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[12] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, 2009.

[13] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86), February 2010. In submission.

[14] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In Atul Prakash and Indranil Gupta, editors, *ICISS*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2009.

[15] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, pages 409–417, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[16] Tzi-cker Chiueh and Manish Prasad. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224. USENIX, 2003.

[17] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing*, pages 196–206, 2007.

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 91–104, Berkeley, CA, USA, 2003. USENIX Association.

[19] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78, Berkeley, CA, USA, 1998. USENIX Association.

[20] Dino Dai Zovi. Practical return-oriented programming. SOURCE Boston 2010, April 2010. Presentation. Slides: http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[21] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on ARM. Technical Report HGI-TR-2010-002, Ruhr-University Bochum, July 2010. Online: http://www.trust.rub.de/home/_publications/DaDmSaWi2010/.

[22] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings of the 4th ACM Workshop on Scalable Trusted Computing (STC'09)*, pages 49–54. ACM, 2009.

[23] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[24] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 15–26, New York, NY, USA, 2008. ACM.

[25] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the 1st Workshop on Secure Execution of Untrusted Code (SecuCode'09)*, pages 19–26. ACM, 2009.

[26] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 55–66, Berkeley, CA, USA, 2001. USENIX Association.

[27] gera. Advances in format string exploitation. *Phrack Magazine*, 59(12), 2002.

[28] Dan Goodin. Apple quicktime backdoor creates code-execution peril. http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/, 2010.

[29] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 65–72, New York, NY, USA, 2006. ACM.

[30] Josh Halliday. Jailbreakme released for apple devices. http://www.guardian.co.uk/technology/blog/2010/aug/02/jailbreakme-released-apple-devices-legal, August 2010.

[31] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp.

[32] Michael Howard and Matt Thomlinson. Windows vista isv security. http://msdn.microsoft.com/en-us/library/bb430720.aspx, April 2007.

[33] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[34] Intel Corporation. Intel 64 and ia-32 architectures software developer's manuals. http://www.intel.com/products/processor/manuals/.

[35] Intel Parallel Studio. http://software.intel.com/en-us/intel-parallel-studio-home/.

[36] Vincenzo Iozzo and Ralf-Philipp Weinmann. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/, Mar 2010.

[37] jduck. The latest adobe exploit and session upgrading. http://blog.metasploit.com/2010/03/latest-adobe-exploit-and-session.html, 2010.

[38] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[39] Tim Kornau. Return oriented programming for the ARM architecture. http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf, 2009. Master thesis, Ruhr-University Bochum, Germany.

[40] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[41] Felix Lindner. Developments in Cisco IOS forensics. CONFidence 2.0. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf, November 2009.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 190–200, New York, NY, USA, June 2005. ACM Press.

[43] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. http://support.microsoft.com/kb/875352/EN-US/, 2006.

[44] Nicholas Nethercote. Dynamic binary analysis and instrumentation. http://valgrind.org/docs/phd2004.pdf, 2004. PhD thesis, University of Cambridge.

[45] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[46] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2005.

[47] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free : defeating return-oriented programming through gadget-less binaries. In *ACSAC'10, Annual Computer Security Applications Conference*, December 2010.

[48] PaX Team. http://pax.grsecurity.net/.

[49] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). *Computer Security Applications Conference, Annual*, 0:60–69, 2009.

[50] Heise Security. Pwn2Own 2009: Safari, IE 8 and Firefox exploited. http://www.h-online.com/security/news/item/Pwn2Own-2009-Safari-IE-8-and-Firefox-exploited-740663.html, 2010.

[51] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[52] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS 04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[53] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702, 2008.

[54] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[55] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. http://www.phreedom.org/research/bypassing-browser-memory-protections/, August 2008. Presented at Black Hat 2008.

[56] SPEC Standard Performance Evaluation Corporation. http://www.spec.org.

[57] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

[58] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, 2010.

[59] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10)*, Oakland, CA, May 2010.