RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

Horst-Görtz Institut
für IT Sicherheit

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
March 19, 2010
ISSN

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 consider the following questions: (i) can any of the existing proposals against memory-based exploits provide an appropriate defense mechanism against ROP attacks? (ii) and if not, can we construct a general and efficient solution based on the extension and improvement of existing techniques? and (iii) if this solution is simple, would it then be of narrow scope and consequently question the relevance of ROP as a "crucial attack"?

We give answers to the first two questions and leave the last question open, although we believe that ROP attacks are real threats and will increasingly be used in the near future and, hence, efficient countermeasures are strongly needed. We present *ROPdefender*, an immediate and practical solution against conventional ROP attacks based on return instructions. It protects arbitrary binary programs without the need to change or inspect their source code. *ROPdefender* consists of only 80 LOC and uses an existing binary instrumentation framework to detect and prevent ROP attacks. In contrast to the well-established countermeasures against memory-based exploits, *ROPdefender* is the first tool that can detect execution of *unintended* instruction sequences even if the adversary has successfully subverted the control flow of a program by other means. Our prototype implementation induces a performance overhead (on Intel x86) on average by a factor of about 2x, which defines a new upper bound for software-based countermeasures of ROP.

1 Introduction

Runtime attacks on software aim at changing the behavior of running programs or taking the control over the underlying machine. Typically, attackers try to inject malicious code in existing programs that can exploit various vulnerabilities of software programs. Buffer overflow vulnerabilities are a prominent example among them (see, e.g., [3]).

A great deal of attention was paid to the $W \oplus X$ security model implemented by the PaX Team [35] that marks a memory page either writable or executable in order to prevent code injection attacks. With $W \oplus X$ the adversary is no longer able to inject malicious code because the injected code has to be placed into some writable (but not executable) memory area.

However, *return-oriented programming (ROP)* [38] 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* [40]. 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.

Is ROP of narrow scope? The power of ROP has been shown on various processor architectures: on PC platforms with Intel x86 [38] or RISC processors such as SPARC [6]; on mobile devices with ARM based architectures [29] such as Apple's smartphone iPhone [27]; and even on Harvard architectures such as AVR microcontrollers [19], or Z80 processors in voting machines [9]. Moreover, Hund et al. [24] presented a return-oriented rootkit for the Windows operating system that bypasses kernel integrity protections. 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.

Do existing countermeasures defend against ROP? Many defense mechanisms against memory-based exploits were proposed in the past: protecting the return address by using compiler extensions [12, 16, 43, 23, 15], binary instrumentation [22, 13], compiler-based taint tracking [30, 44], taint tracking based on binary instrumentation [34, 14], or control flow integrity [1, 2]. These countermeasures mainly aim at preventing the adversary from subverting the control flow of a program. However, this goal is a hard task (if not impossible) as long as software is written in unsafe languages like C/C++. Therefore we assume a (strong) adversary model that allows to subvert the control flow of a program. To the best of our knowledge, under this assumption, none of the mentioned countermeasures can detect execution of *unintended* instruction sequences issued during a ROP attack.

Even defense mechanisms such as XFI [2] (which is based on control-flow integrity [1]) or return address protection schemes using binary instrumentation [13, 22] fail, because they only enforce return address checks on *intended* instruction sequences such as function prologues and epilogues. XFI might rule out many ROP attacks by enforcing control-flow integrity. But if it is possible to execute an unintended instruction sequence in an XFI-protected program, then no instrumented runtime check would detect this and, hence, the adversary could launch a ROP attack. Hence, the failure mode of XFI is much more dangerous compared to *ROPdefender*, because XFI cannot detect the execution of unintended instruction sequences. Moreover, XFI is based on Microsoft Vulcan's instrumentation framework, which is not publicly available and restricted to the Windows operating system. Further, XFI relies on debugging information stored in Windows PDB files. Such files are not provided by default for each application. Hence, hand-written libraries are not supported by the rewriter. In contrast, *ROPdefender* needs no extra information on the program's structure, can be applied to any binary, and is based on the publicly available Pin framework [31].

Further, hardware-facilitated solutions such as StackGhost [21] using stack cookies can also not detect any kind of ROP attack [6]. The ROP attack will be still successful if the adversary initiates it by means of a format string [37] or a heap overflow [4] instead of corrupting a return address. However, Frantzen et al. [21] also propose a return address stack, but to the best of our knowledge, this has never been implemented.[1] Additionally, StackGhost is only available on SPARC, where *unintended* instruction sequences cannot be issued due to aligned memory access and fixed instruction length. Finally, StackGhost depends on SPARC's register window overflow and underflow traps, which is a unique feature on SPARC systems.

Our contributions. We present the design and implementation of *ROPdefender*, a tool that is able to detect/prevent ROP attacks. We make use of known techniques such as *instrumentation*, particularly instrumentation based on a just-in-time (jit) compiler, which adds extra code to a program's binary at runtime with the purpose to observe the program's behavior during execution. In contrast to existing countermeasures against memory-based exploits, our tool can even detect execution of *unintended* instruction sequences when the adversary is able to tamper with the control flow. 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 [12, 43, 21] for return addresses, and the concept of binary instrumentation as used in taint tracking [34, 14] or return address protection [22, 13]. To the best of our knowledge, *ROPdefender* is the first return address protection system using jit-based instrumentation to enforce return address checks. Our contributions are summarized as follows:

- *ROPdefender* requires *no access to the source code* of programs while enforcing instrumentation on their binaries. Contrary to XFI [2] *ROPdefender* does also not depend on debugging information stored in PDB files.

[1] http://projects.cerias.purdue.edu/stackghost/

- For this purpose we use the jit-based binary instrumentation framework Pin [31]. Typically, Pin is used for program analysis such as performance evaluation and profiling, taint analysis [14], or memory and thread checking as in Intel Parallel Studio [26]. Since many taint analysis systems are based on binary instrumentation, we believe that *ROPdefender* can be easily incorporated into other vulnerability detection frameworks.

- We address ROP attacks on the Intel x86 architecture where unintended instruction sequences can be issued. However, *ROPdefender* can be easily adopted to other processor architectures since the solution is generally applicable and the underlying Pin framework can be deployed to various processor architectures.

- Contrary to prior work, *ROPdefender* is able to detect execution of *unintended* instruction sequences, specifically unintended return instructions. This is achieved by enforcing a return address check on any return instruction issued during program execution. However, *ROPdefender* also prevents all common return address corruption attacks such as conventional buffer overflows [3] or return-into-libc attacks [40].

- *ROPdefender* provides strong protection against ROP attacks based on return instructions. However, recently Checkoway and Shacham [10] presented a new ROP attack that only make use of indirect jumps instead of return instructions. We will discuss in Section 7 if it is possible to detect such attacks with *ROPdefender*.

Our proposed instrumentation technique is to the best of our knowledge the first software-based detection method against general ROP attacks. Recently, Chen et al. proposed DROP [11] a binary instrumentation tool to defeat ROP attacks. As we will see in Section 6 on related work, a ROP adversary is able to bypass DROP with low effort. DROP only targets ROP attacks occurring in libc, although it induces a performance overhead on average by a factor of 5.3x. In contrast, *ROPdefender* targets all ROP attacks during program execution and induces only a performance overhead on average by a factor of 2.17x (integer computation) and 1.41x (floating point computation).

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. In Section 7 we discuss ROP attacks that are based on indirect jumps and conclude the paper in Section 8.

2 Background on Return-Oriented Programming (ROP)

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 ROP Attack

The main goal of a conventional buffer overflow attack [3] 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.

If the $W \oplus X$ model [35] 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* [40].

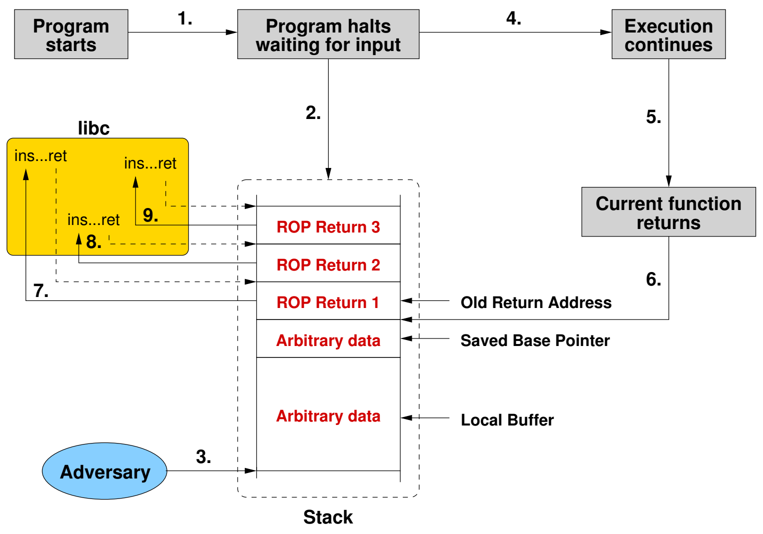

Figure 1: Return-Oriented Programming Attack

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 [38], called return-oriented programming (ROP). ROP [38] can be seen as generalization of return-into-libc attacks that resolves the constraints of traditional return-into-libc attacks. With ROP arbitrary computation can be performed without injecting new code and without calling any functions. ROP attacks are even applicable for established systems such as SPARC [6], Harvard architectures [19], voting machines [9], and even on ARM based architectures used in mobile devices [29].

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 1 depicts a general ROP attack initiated by a buffer overflow. The difference to a conventional buffer overflow [3] 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.

In step 1 the vulnerable program is started by an authorized user. After the program is initialized, user's input is expected by the program that will afterwards be stored in a local buffer on the stack (step 2). The adversary inserts longer input than expected by the program, whereas the input of the adversary consists of arbitrary data (to overfill the local buffer and to overwrite the saved base pointer) and various return addresses that point to instruction sequences in libc (or any other existing code linked to the program). Thus, in step 3, the original return address is overwritten with the start address (ROP Return 1) of the first instruction sequence. After retrieving user input, the vulnerable function continues execution until a return instruction is issued (steps 4-6). Then the processor executes the return instruction and redirects control to the first instruction sequence (step 7). Upon return of the first instruction sequence, control transfers to the second instruction sequence pointed to by ROP

[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.

Return 2 and then the steps 8 and 9 are executed similarly to step 7.

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 [25] is a well-established instruction set architecture deployed in personal computers. Shacham [38] 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 [38] especially makes use of unintended instruction sequence ending in a return instruction to enforce ROP attacks.

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. *No source code available:* We assume that we have no access to the source code while defeating ROP. First, source code is often not available and second, some libraries are directly written in the assembly language. Moreover, compiler-based solutions require a recompilation process.

2. *Strong adversary:* The adversary is able to launch the first instruction sequence in a ROP attack. Hence, the adversary is able to subvert the control flow of a program, e.g., by attacks such as buffer overflow, heap overflow, format string, etc. Thus, the adversary is able to push various return addresses on the stack in order to issue a ROP attack.

3. *Integrity of ROPdefender:* The adversary cannot attack the *ROPdefender* software 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 *ROPdefender*.

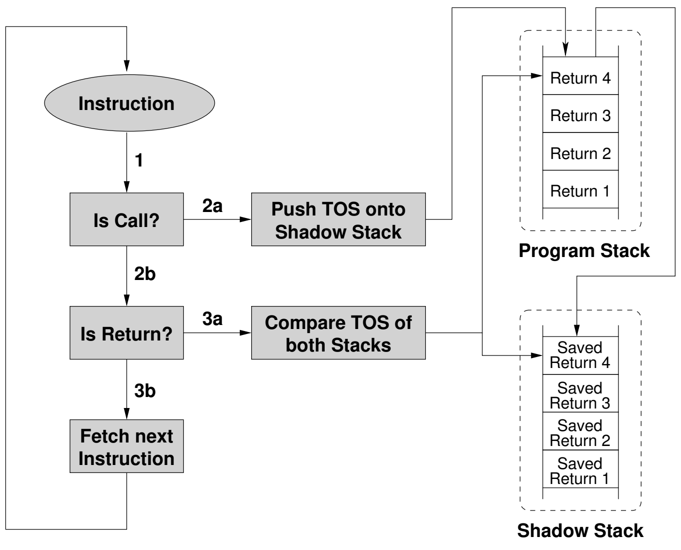

Figure 2: Our high-level approach to defeat ROP

3.2 High-Level Idea

Basically, when the program execution has to be transferred to a subroutine, the call instruction itself pushes the return address onto the stack. Afterwards, the called subroutine completes its task and usually returns to its original caller. However, there are a few exceptions that break the traditional calling convention and the function returns elsewhere. We will discuss these exceptions in Section 3.4. For the moment, we assume that a function returns always to the return address originally pushed by the call instruction on the top of the stack (TOS). Nevertheless, our prototype implementation of *ROPdefender* also handles the exceptions (see Section 4).

ROP attacks misuse the task of the stack pointer. In ordinary programs, upon return of a function, the stack pointer will point to the return address so that control transfers back to the calling function. As described in Section 2, the adversary misuses the return instructions at the end of each instruction sequence in order to transfer control to the subsequent instruction sequence. The return instruction can be an intended instruction originally placed by the programmers of libc, or an unintended instruction where the byte value of the return instruction is just a suffix of another valid instruction.

For this reason we particularly focus on call and return instructions as depicted in Figure 2. In order to evaluate each return instruction issued during program execution, we store a copy of the return address onto a separate *shadow stack* (similar to [12, 43, 21]) once the program issues a call instruction. We instrument all return instructions that are issued during program execution and perform a return address check as described in the following:

1. Before an instruction will be executed by the processor, our solution intercept the instruction and evaluates the instruction's type and target. In practice, this can be accomplished with a binary instrumentation framework as we will explain in Section 3.3 and 3.4.

2. First, we check if the instruction is a call instruction. If this is the case, we store a copy of the pushed return address onto our shadow stack (transition 2a in Figure 2). Otherwise, we check whether the intercepted instruction is a return instruction (transition 2b in Figure 2).

3. If the instruction is a return instruction, our solution checks if the top return address on the shadow stack equals the return address on top of the program stack. If the two return addresses are different, we conclude that a return address corruption has occurred and program execution is redirected to instructions not intended by the programmer.

Our solution prevents traditional buffer overflow attacks that are launched by modifying a return address. But it also provides detection of ROP attacks that use unintended instruction sequences. After the adversary issues the first instruction sequence in a ROP attack, the return instruction at the end of the first instruction sequence will transfer control to the second instruction sequence. At that time, our solution will enforce a return address check as depicted in Figure 2 step 3a.

The program stack will hold the start address of the second instruction sequence. However, since our shadow stack holds only return addresses that are pushed by call instructions themselves, it cannot contain the start address of this instruction sequence starting somewhere in the middle of a function. Thus, any return address violation can be detected by our solution.

Obviously, in our approach we assume that instruction sequences issued during a ROP attack end in return instructions. Actually, that is the original ROP attack presented by Shacham [38]. One idea to bypass our solution is to use instruction sequences ending in an indirect jump/branch instruction. Actually, Shacham briefly mentions the ability of using indirect branch instructions and recently published a paper [10] that describes such attacks for Intel x86. These attacks are beyond the scope of this paper. However, we will discuss in Section 7 how such attacks can be detected.

In Section 3.4 we will describe how our solution can be implemented into a binary instrumentation framework like Pin.

3.3 Tools and Techniques

We use a technique, which is called *instrumentation*, that adds extra code into a program to observe and debug the program's behavior [32]. Generally, instrumentation can be performed at runtime, at compile-time, or within the source code. Since we assume (see Section 3.1) no access to the source code, we focus on dynamic binary instrumentation at runtime. Generally, there are two classes of dynamic binary instrumentation frameworks: (i) probe-based instrumentation frameworks and (ii) instrumentation frameworks that uses a just-in-time compiler (jit-based).

Probe-based instrumentation used for instance in DynInst [7], Vulcan [18] and DTrace [8] enforce instrumentation by replacing instructions with so-called trampoline instructions that branch to instrumentation code. DTrace, for instance, replaces instrumented instructions with special trap instructions that once issued generate an interrupt. When the interrupt occurs, the instrumentation code and the replaced instruction are executed.

Jit-based instrumentation frameworks like Valgrind [33], DynamoRIO [5], and Pin [31] use a just-in-time compiler. Contrary to probe-based instrumentation no instructions nor bytes in the executable are actually 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.

In order to defeat ROP attacks that are based on unintended instruction sequences, we use jit-based instrumentation since they perform better than probe-based instrumentation for the following reason: With jit-based instrumentation we are able to intercept each instruction before it is executed by the processor, whether the instruction was intended by the programmer or not. Probe-based instrumentation frameworks rewrite instructions ahead of time with trampoline instructions and therefore instrumentation is only performed if the trampoline instruction is really reached during program execution.

3.4 General Architecture

The architecture of *ROPdefender* consists mainly of two components, *ROPdefender* detection unit and a binary instrumentation engine. Both components, as well as the operating system, are part of our trusted computing base (TCB). Figure 3 depicts our proposed architecture to effectively defeat ROP attacks.

Binary Instrumentation Framework. In our architecture we use a jit-based instrumentation framework for the purpose of detecting ROP attacks. Typically, a jit-based instrumentation framework consists of a virtual machine (VM) and a JIT compiler. The general workflow is as follows: The program is loaded and started under the control of the instrumentation engine. The first instruction is intercepted by the engine and the JIT compiles the instruction into new instructions in order to incorporate instrumentation code according to the *ROPdefender* detection unit. Therefore, the detection unit connects to the instrumentation engine via interface I1. The compiled instructions are sent to the operating system, i.e., finally to the CPU, through interface I2. Then, these steps are repeated for all subsequent instructions of the program binary.

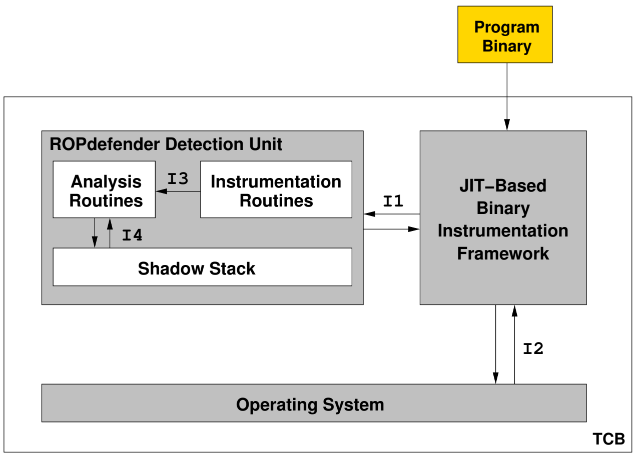

Figure 3: General architecture of *ROPdefender*

***ROPdefender* Detection Unit.** Our *ROPdefender* detection unit consists of instrumentation and analysis routines, and a shadow stack. Instrumentation routines usually inspect the current instruction in general (e.g., check if the current instruction is a call, a return, a branch, etc.). The instrumentation routines are able to call different analysis routines through interface I3 based on the type of the current instruction. For instance, if the instruction is a call instruction, an analysis routine can be invoked that performs further analysis of the call instruction. Is the current instruction instead a return instruction, then a second analysis routine can be called performing analysis different from the call analysis. The detection unit also includes the shadow stack. Analysis routines can push and pop return addresses onto and off the shadow stack through interface I4.

4 Implementation

In this section we describe implementation details of our framework and the *ROPdefender*. For our implementation we used the jit-based binary instrumentation framework Pin in version 2.6-27887. Further, our implementation of the *ROPdefender* detection unit is one C++ file consisting of 80 lines of code. Finally, we will also discuss how our *ROPdefender* can handle exceptions such as non-local control transfers.

Pin achieves the best performance among jit-based instrumentation frameworks [31]. Pin is typically used for program analysis such as performance evaluation and profiling.[3] Intel uses Pin in the Intel Parallel Studio [26] for memory and thread checking or bottlenecks determination. However, we use this binary instrumentation framework for the purpose of detecting ROP attacks. Figure 4 shows the instantiation of our architecture with the Pin framework.

Pin. Pin itself has mainly two components: (i) a code cache and (ii) a Virtual Machine (VM) whereas the VM contains the JIT compiler and an emulation unit. Pin is configured via so called 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* with instrumentation and analysis routines.

After Pin is loaded and initialized, Pin initializes the *ROPdefender* detection unit. Then the program which we want to protect is started under the control of Pin. The first instruction is intercepted by Pin and the JIT compiles the instruction into new instructions to incorporate instrumentation code. The compiled code is then transferred to a code cache by 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.

[3]Although DYTAN [14] uses Pin to enforce dynamic taint analysis.

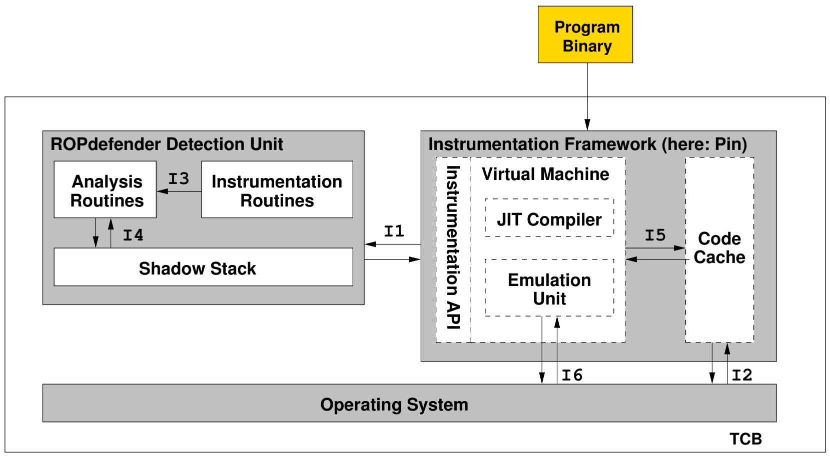

Figure 4: Implementation of *ROPdefender* architecture using the binary instrumentation framework Pin

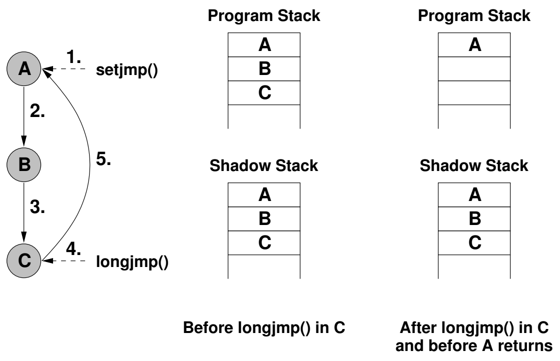

Figure 5: Call Path for setjmp and longjmp

Instrumentation and Analysis Routines. According to Figure 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 performs the actions and checks according to the steps 2a and 3a. To implement the shadow stack we additionally use the C++ stack template container that operates like a shadow stack in the application. 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* uses the inspection routines *Ins_IsCall(INS ins)* and *Ins_IsRet(INS ins)* provided by the Pin API to determine if the intercepted instruction is a call or a return instruction. If the instruction is a call instruction, then we invoke an analysis routine (step 2a) that is responsible for pushing the return address onto our shadow stack. Otherwise, if the instruction is a return instruction, then a second analysis routine enforces a check between the return address the stack pointer points to (i.e., the return address on the program stack) and the saved return address placed on top of the shadow stack according to step 3a.

Exceptions. As mentioned above, usually we can assume that an invoked function will always return to the address pointed to by the return address pushed onto the stack by the calling function. However, there are some exceptions: 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 and broke therefore the usual calling convention with a non-local control transfer that can be seen as a non-local goto. Figure 5 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.

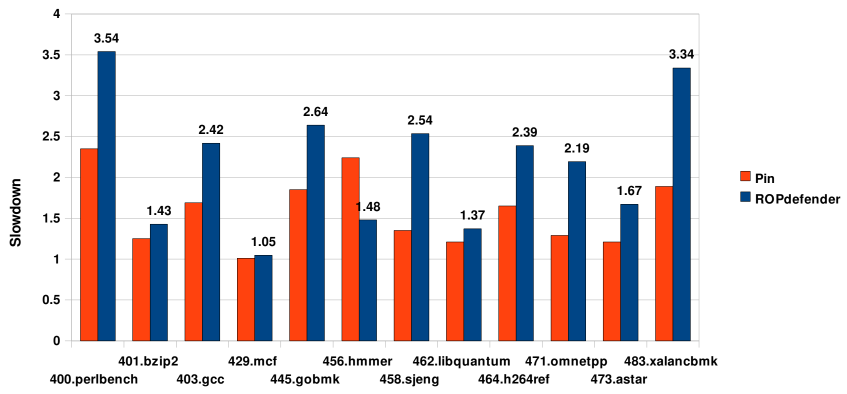

Figure 6: **SPEC Integer Benchmark Results**: The Figure depicts the slowdown of applications running under Pin but without instrumentation and running under Pin with *ROPdefender*. The data labels depict only the precise slowdown values for our *ROPdefender*.

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 eliminate this false positive, *ROPdefender* uses strategies as, for instance, implemented in RAD [12] that pops 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 or a traditional return address corruption.

5 Evaluation

5.1 Performance

This Subsection presents the performance overhead introduced by our architecture with the binary instrumentation framework Pin and our *ROPdefender*. We compare our results with programs running under no instrumentation and running under Pin but without instrumentation. Our testing environment was a 3.0 GHz Intel Core2 Duo E6850 machine running Ubuntu 9.04 (i386) with Linux kernel 2.6.28-11 and Pin version 2.6-27887. We ran the integer and floating-point benchmarks from the SPEC CPU2006 Suite [41] using the reference inputs.

Pin without Instrumentation. First, we measured the performance overhead the Pin binary instrumentation framework itself adds to the running applications. The average slowdown is 1.58x for integer computations and 1.15x for floating point computations. The slowdown for integer computations ranges form 1.01x (429.mcf) up to 2.35x (400.perlbench). In contrast, for floating point computations the slowdown ranges from 1.004x (470.lbm) up to 1.57x (447.dealII).

Pin with *ROPdefender*. Applications under protection of our *ROPdefender* run on average 2.17 times for integer and 1.41 times for floating point computations slower than applications running without Pin. The slowdown for the integer benchmarks ranges from 1.05x up to 3.54x and ranges from 1.01x up to 3.06x for the floating point benchmarks. Our *ROPdefender* adds a performance overhead of 1.37x for integer and 1.19x for floating point computations in average compared to applications running under Pin but without instrumentation. However, the performance overhead introduced by dynamic taint analysis tools such as DYTAN [14] (also based on Pin) or TaintCheck [34] (based on Valgrind) is enormously higher. Applications running under these tools are from 30x up to 50x times slower. Also DROP [11] induces a higher performance overhead than *ROPdefender*. The slowdown is on average 5.3x.

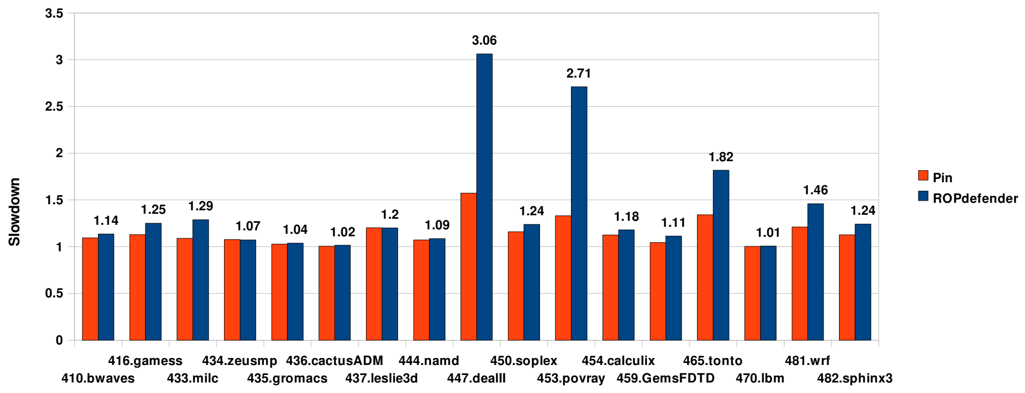

Figure 7: **SPEC Floating Point Benchmark Results**: The Figure depicts the slowdown of applications running under Pin but without instrumentation and running under Pin with *ROPdefender*. The data labels depict only the precise slowdown values for our *ROPdefender*.

5.2 Security Analysis

In the following we show that our *ROPdefender* is able to detect and prevent sophisticated ROP attacks. Figure 8a) illustrates a common calling sequence in an ordinary program. Once the main routine calls the function A, a new stack frame is established with input arguments (without loss of generality, Arg 1 and Arg 2). The return address of function A and a local data buffer are directly placed below these arguments.[4] After function A completes its task, it returns back to the main routine as indicated by the return address.

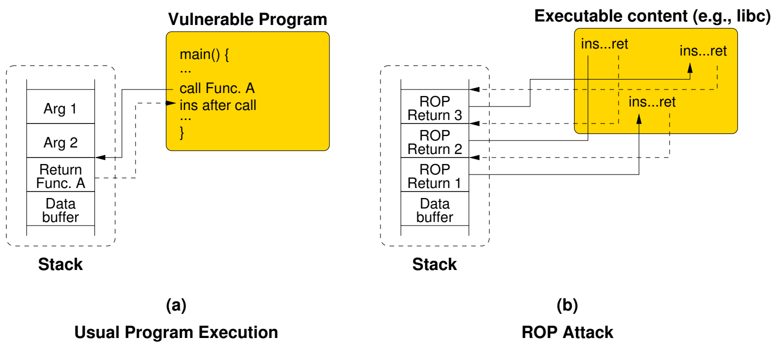

Figure 8: A general ROP attack

The general ROP attack depicted in Figure 8b) shows the situation where the adversary was able to push (e.g., due to a buffer overflow attack) various return addresses onto the stack that each points to instruction sequences in the executable memory regions. Instruction sequences can be intended or unintended ones, but they always end in a return instruction in order to transfer control to the subsequent instruction sequence.

As we described in Section 4, our *ROPdefender* stores a copy of the return address at function entry on a separate shadow stack. If no further call instructions occur in function A, the shadow stack will remain unchanged until the return instruction of the function has been reached. However, we assume that the adversary is able to subvert the control flow (see Section 3.1). Hence, we assume that control transfers to an instruction sequence outside function A before the original return instruction of A has been reached.

[4]The saved base pointer is usually placed between the data buffer and the return address, which we ignore for the moment for brevity reasons. However, usually the adversary simply overwrites the saved base pointer with arbitrary data.

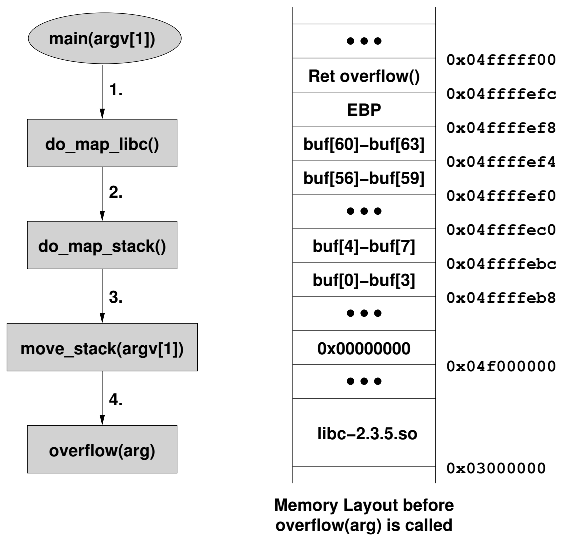

Figure 9: Structure of vulnerable program with corresponding memory layout

For instance, this can be achieved by corrupting a function pointer or using a format string vulnerability. The first instruction sequence is then executed which always ends in a return instruction. Since the binary instrumentation framework (Pin) assures that each return instruction issued during program execution will be instrumented and *ROPdefender* enforces a check between the return addresses on the program stack and the shadow stack, the ROP attack can be detected: the address stored on the shadow stack is the return address of function A and the return address on the program stack is the start address of the second instruction sequence.

ROPdefender is able to detect all ROP attacks because our jit-based instrumentation framework instruments each instruction during the execution of the program. Hence, all instructions, whether they were originally intended or not, are intercepted by our jit-based instrumentation framework before they will be executed by the processor as described in Section 3.4. Since ROP attacks are based on return instructions placed at the end of each instruction sequence, we are able to catch all return instructions issued in a ROP attack with our instrumentation routine *Ins_IsRet(INS ins)*. This function even instruments all return instruction variants as far return instructions and return instructions with stack unwind.

In contrast to probe-based instrumentation frameworks, jit-based instrumentation does not change the binary. Instead, instructions are intercepted at runtime (as we described in Section 3.4) and therefore we are able to catch and instrument instructions that were never intended by the programmer. Shacham [38] uses particularly such unintended instructions sequences on the x86 architecture.

Due to our assumption 3 in Section 3.1 the adversary is not able to disable our instrumentation framework with the corresponding *ROPdefender*, because both are included in our TCB. Therefore an instruction issued during program execution cannot bypass our instrumentation framework and the *ROPdefender*.

Example. To illustrate the general defense mechanism of *ROPdefender* with an example, we show in detail that our *ROPdefender* is able to detect the practical ROP attack presented in the full paper of Shacham[5] [38]. The depicted example program suffers from a buffer overflow vulnerability and allows a ROP attack to occur. The target of the ROP attack is to launch a shell to the ROP adversary. Figure 9 depicts the structure and the corresponding memory layout of the vulnerable program. In the following we will describe which commands are issued within the depicted functions and explain how the corresponding memory layout is achieved.

1. First the main routine in the vulnerable program is started with an input parameter inserted by the

[5]The full paper is an extended version of the published paper by Shacham [38] and is available on `http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf`

user (i.e., the adversary) on the command line. The payload of the adversary consists of arbitrary data to overfill a local buffer and of return addresses whereas each points to an instruction sequence in libc. Afterwards, the main routine calls the *do_map_libc()* function that maps the UNIX libc library to the address `0x03000000`.

2. The second function called by the main routine is the *do_map_stack()* function. This function opens the special file */dev/zero* and maps it to the address `0x04f000000`. Since the */dev/zero* file provides only zero values, the memory area starting from `0x04f000000` up to `0x050000000` will also contain only zero values. As we will describe in the next item, this memory area will be used for establishing new stack frames.

3. The *move_stack(argv[1])* function changes the stack pointer to the address `0x04ffff00`. Therefore the stack pointer points to the memory area initialized in step 2. Further, the *move_stack(argv[1])* calls the *overflow(arg)* function. The return address of the *overflow(arg)* function is pushed at address `0x04ffffefc`.

4. This function initializes a local 64 byte buffer and afterwards uses the dangerous *strcpy(buf,arg)* function to copy the user input (i.e., the adversary's payload) into the local buffer without enforcing bounds checking.

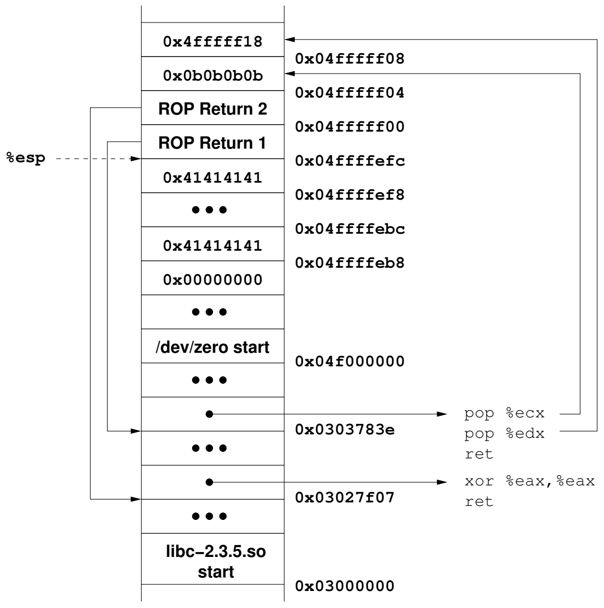

Figure 10: Memory layout before first instruction sequence of ROP attack is issued

The adversary can, for instance, initiate the ROP attack by means of a conventional buffer overflow attack. 64 arbitrary bytes (e.g., 'A' (0x41) characters) are needed to fill the local buffer and four further bytes will overwrite the saved base pointer that is stored directly above the last buffer element (buf[63]). The next bytes in the payload are return addresses and in between some arguments that are needed for some instruction sequences.

Figure 10 depicts the memory layout after *strcpy(buf,arg)* was invoked whereas we depict for brevity reasons only the first two return addresses and the first two arguments from the original payload.

Figure 11 depicts return addresses on the program and shadow stack before and after the buffer overflow. In Figure 11a) both stacks are equal. However, the ROP adversary pushes various return addresses on the program stack and overwrites the original return address of *overflow()* with the address (`0x0303783e`) which is start address of the first instruction sequence in libc. Upon return of the *overflow()* function, *ROPdefender* will detect a mismatch since both TOS are not equal which is a clear indication for a return address corruption.

However, the adversary might be able to initiate the ROP attack not by means of a buffer overflow and by not corrupting the return address of the *overflow()* function. For instance, he could be able

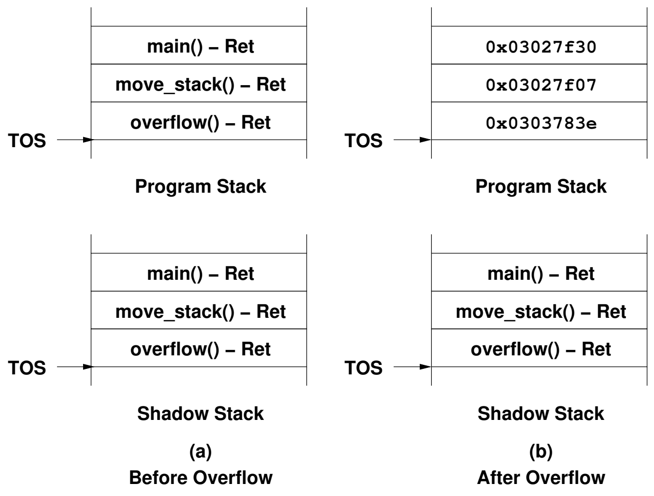

Figure 11: Return addresses on program and shadow stack before and after buffer overflow attack

to corrupt another function pointer that transfers control to the first instruction sequence starting at `0x0303783e`. In such a case the first instruction sequence (`xor %eax,%eax; ret;`) will be executed. We further assume that the first instruction sequence ends in an unintended return instruction. Upon return of the first instruction sequence, the TOS of the shadow stack will still hold the return address of *overflow()*. But the top of the program stack holds the start address (`0x03027f07`) of the second instruction sequence. Since *ROPdefender* is able to enforce a return address check on unintended return instructions, it will recognize the mismatch and therefore is able to detect the ROP attack although the adversary used unintended instructions and was able to transfer control to the first instruction sequence.

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.

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.

The $W \oplus X$ Model. This protection scheme prevents conventional buffer overflow attacks that redirects 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 [35]. 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.

Address Space Layout Randomization (ASLR). PaX Address Space Layout Randomization (ASLR) [35] was proposed to prevent return-into-libc attacks by randomizing the base addresses of program's text segments. As the adversary has to know the precise address of a libc function, this approach seems to effectively prevent return-into-libc and also ROP attacks. However, there exists derandomization techniques to bypass ASLR [39].

Compiler Extensions. Since return addresses are the most common target for traditional buffer overflow attacks on the stack, a few approaches were proposed that aim to detect malicious change of return addresses, whereas most of them are implemented as compiler extensions. The famous StackGuard compiler [16] places a dummy value, referred to as canary value, 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. Only if the canary value is valid, the return to the caller is permitted. In contrast, Stack Shield [43] guards the return address by introducing a second stack, a Global Ret Stack, that holds a copy of return addresses. At function entry the return address is pushed onto the Global Ret Stack and copied back upon function return. A very similar approach is proposed by Chiueh et al. [12], called Return Address Defender (RAD), that stores return addresses into a safe memory area. ProPolice [23] 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 be not overwritten. Only the return address and the saved base pointer are overwritten which are still protected by the guard value. A more general approach, called PointGuard [15], encrypts all pointers and only decrypts them when they are loaded into CPU registers. Registers cannot be directly accessed by the adversary and are therefore not vulnerable to buffer overflow attacks. Hence, the adversary has only access to encrypted pointers stored on memory.

However, all these mitigation techniques require recompilation and hence access to the source code. Further, none of them is able to detect the execution of unintended instruction sequences if the adversary successfully subverted the control flow of the program. Nevertheless, in our approach we use the idea of keeping a copy of the return address onto a shadow stack as used in Stack Shield [43].

Return Address Protection without access to the Source Code. A few approaches were proposed (see, e.g., [22, 13]) that aim to detect malicious change of return addresses by using instrumentation and rewriting techniques without needing access to source code. Both approaches rewrite function prologue and epilogue instructions with a branch instruction to code that stores a copy of the return address and enforce a return address check on function return. Chiueh et al. [13] use static instrumentation which means that the binary is disassembled and rewritten before it is executed. But accurate disassembly ahead of time is difficult to achieve and error-prone [13]. Gupta et al. [22] use probe-based instrumentation to rewrite function prologue and epilogue instructions. However, both approaches are not able to detect ROP attacks that use *unintended* instruction sequences, because only intended instructions are instrumented.

Dynamic Taint Analysis. Dynamic taint analysis marks any untrusted data 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 during program execution, and alerts or terminates the program if tainted data is misused. Misuse of the tainted data is, for instance, using the tainted data as a pointer, since most buffer overflow attacks are based on changing return addresses and function pointers. Taint analysis systems can be compiler-based (see, e.g., [30, 44]) or based on binary instrumentation (wee, e.g., [34, 14]).[6]

Since our solution should not depend on access to the source code, only taint analysis based on binary instrumentation could be used in our defense mechanism. Taint analysis systems based on binary instrumentation use a binary instrumentation framework and focus on preventing traditional runtime attacks as buffer overflows by preventing the adversary from subverting the control flow. Effectively, our solution also uses a binary instrumentation framework. We believe that *ROPdefender* can be incorporated into existing taint analysis systems. However, to the best of our knowledge, we are the first using binary instrumentation to detect the execution of unintended instruction sequences after the adversary successfully subverted the control flow.

Control Flow Integrity (CFI). 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 the beginning of a particular function but does not affect the behavior of the function.[7] 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.

[6] Actually, also hardware-based taint analysis is available. However, we focus on software-based defense mechanisms.
[7] They are similar to a nop instruction. Abadi et al. [1] propose the prefetchnta instruction.

XFI's rewriting engine first disassembles the binary in order to find all branch instructions (such as return instructions) and afterwards rewrites the return instructions with additional instrumentation code to enforce 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, 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 [18] used by XFI is not publicly available and is restricted to the Windows operating system. Moreover, to build the CFG, XFI requires some 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.

CPU-based protection. A hardware-based solution to defend against ROP attacks is proposed in [20]. In that approach an embedded microprocessor is adapted to include memory access control for the stack. Moreover, the stack is split into one for data and one only for call and return addresses. The processor implements access control mechanisms that do 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 [21]. StackGhost is based on stack cookies that are XORed against return addresses at function entry and XORed out upon function return. Frantzen et al. [21] also propose an alternative solution that is based on a return address stack (similar to our shadow stack), but to the best of our knowledge, this was never implemented. Further, StackGhost depends on specific features, which are unique to SPARC, such as the register window underflow and overflow traps.

DROP: Detecting ROP Malicious Code. Chen et al. [11] recently introduced DROP, a binary instrumentation tool to defeat ROP. DROP aims to detect ROP attacks by counting instructions issued between two return instructions. If such short instruction sequences are invioked three times in a row, DROP reports a ROP attack. To bypass DROP, a ROP adversary could enlarge the instruction sequences or enforce a longer instruction sequence after, for instance, each second instruction sequence. Therefore, DROP provides no general detection of ROP. Moreover, DROP only analyzes return instruction issued in libc. Contrary, *ROPdefender* instruments any return instruction issued during program execution and *ROPdefender* induces less performance overhead than DROP (see Section 5).

Trusted Computing The Trusted Computing Group (TCG) is an industrial initiative towards the realization of Trusted Computing. A special hardware security module called *Trusted Platform Module* (TPM) [42] can be embedded in computer mainboards to provide some cryptographic functions and protected storage for cryptographic keys. Trusted Computing provides a technique called *attestation* that reports the system state to a (remote) party. For instance, the Integrity Measurement Architecture (IMA) [36] uses a TPM to record measurements of any dynamic executable content at load-time from the BIOS all the way up to the application layer. However, IMA lacks from providing runtime integrity and can therefore not detect ROP attacks. Nevertheless, recent proposals as ReDAS [28] and DynIMA [17] showed that the TPM can also be used to detect runtime attacks. However, ReDAS requires access to the source code of program binaries and does not address ROP attacks. DynIMA proposes to include instrumentation techniques into runtime attestation to detect ROP attacks, but it is only a position paper and does not provide a working solution. Similar to DROP [11], DynIMA uses a frequency measurement unit in order to detect ROP attacks and suffers therefore from the same shortcomings as DROP does. However, we believe that *ROPdefender* can be integrated into existing runtime attestation tools.

7 Discussion

Checkoway and Shacham [10] recently presented a new ROP attack that is only based on indirect jump instructions. Each instruction sequence ends with an indirect jump that transfers execution to the subsequent instruction sequence. The start addresses of the several instruction sequences are popped off the stack into general purpose registers. To simulate a whole return instruction, they make use of a pop-jump sequence, such as: `pop  %edx; jmp *(edx)`. Unfortunately, they found no such sequence (even not unintended) in their tested libc version. Therefore they introduced the *bring your own pop-jump (BYOPJ)* paradigm. The BYOPJ paradigm assumes that a pop-jump sequence is available in the target program or in one of its libraries. Actually, all instruction sequences end in an indirect jump to this pop-jump sequence, such as: `jmp *x` $\Rightarrow$ `pop y; jmp *y`, where register `x` contains the address of the pop-jump sequence and register `y` holds the address of the next instruction sequence to be executed after the pop instruction.

Certainly, a ROP attack based on indirect jumps cannot be defeated by the current implementation of *ROPdefender*. This is because the attack does not use return instructions at all and thus does not disrupt the calling sequence of the target program. Corruption of return addresses can be detected by *ROPdefender* since the call instruction itself pushes the return address onto the stack at function entry. But without having some information about the program's structure, it seems impossible to decide at runtime if a jump target is a legal one or not. 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 [10], pop-jump sequences are uncommon in ordinary programs, but the ROP attack presented in [10] invokes such a sequence after each instruction sequence. So it might be possible to extend *ROPdefender* with a frequency measurement unit (as proposed in [17] and [11]). 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.[8] The concrete implementation presented in [10] 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 [10] and thus bypass a solution that looks only for returns and indirect jumps. It remains open if an effective countermeasure against ROP can be deployed without knowing the structure of the target program. However, *ROPdefender* with an extended frequency measurement unit rules out 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.

8 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 countermeasure against ROP attacks based on return instructions without access to any source code or dependence on new hardware features. In this paper we presented our *ROPdefender* that fulfills accurately these requirements and that is able to detect/prevent ROP attacks that even are based on *unintended* instruction sequences. For this, we exploited the idea of duplicating return addresses onto a shadow stack and the concept of binary instrumentation to evaluate each return instruction during program execution.

Though we presented an effective and new upper bound for countermeasures on ROP, we believe that ROP attacks still are relevant in today's computing platforms, particularly as soon as $W \oplus X$ will be deployed. *ROPdefender* induces a performance overhead by a factor of 2x which cannot be expected by time-critical applications. However, we believe that performance overhead could be decreased by incorporating our approach in the CPU hardware. Moreover, we need protection against ROP attacks targeting the operating system that *ROPdefender* relies on. But, *ROPdefender* is already a practical solution that can be immediately deployed to protect applications against general ROP attacks. Currently, we are working on a countermeasure against ROP attacks that use indirect jumps instead of return instructions.

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

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] Martín Abadi, Mihai Budiu, Úlfar 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] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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 tzi Advantage. In *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, 2009.

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

[11] 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.

[12] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. *Distributed Computing Systems, International Conference on*, 0:409–417, 2001.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] Vincenzo Iozzo and Charlie Miller. Fun and Games with Mac OS X and iPhone Payloads. In *Black Hat Europe*, Amsterdam, April 2009.

[28] Chongkyung Kil, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, 2009. to appear.

[29] 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.

[30] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472. IEEE Computer Society, 2006.

[31] 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.

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

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

[34] 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.

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

[36] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, 2004.

[37] Scut/team teso. Exploiting format string vulnerability.

[38] 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.

[39] 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.

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

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

[42] Trusted Computing Group. TPM main specification. Specification Version 1.2 rev. 103, July 2007. https://www.trustedcomputinggroup.org/specs/TPM/.

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

[44] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.