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

Lucas Davi[†], Ahmad-Reza Sadeghi[†], Marcel Winandy[‡]

[†]System Security Lab
Technische Universität Darmstadt
Darmstadt, Germany

[‡]Horst Görtz Institute for IT-Security
Ruhr-Universität Bochum
Bochum, Germany

ABSTRACT

Modern runtime attacks increasingly make use of the powerful return-oriented programming (ROP) attack techniques and principles such as recent attacks on Apple iPhone and Acrobat products to name some. These attacks even work under the presence of modern memory protection mechanisms such as data execution prevention (DEP). In this paper, we present our tool, *ROPdefender*, that dynamically detects conventional ROP attacks (that are based on return instructions). In contrast to existing solutions, *ROPdefender* can be immediately deployed by end-users, since it does not rely on *side information* (e.g., source code or debugging information) which are rarely provided in practice. Currently, our tool adds a runtime overhead of 2x which is comparable to similar instrumentation-based tools.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

return-oriented programming, detection, binary instrumentation

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. Typically, the control-flow of a program is subverted by exploiting memory vulnerabilities. Despite extensive research and many proposed solutions in the last decades, such vulnerabilities (e.g., stack overflow [4], heap overflow [5], integer overflow [6], format string [27]) are still the main source of vulnerabilities in today's applications. Figure 1 shows that the number of buffer overflow vulnerabilities (according to the NIST[1] Vulnerability database) continues to range from 600 to 700 per year.

Operating systems and processor manufactures provide solutions to mitigate these kinds of attacks through the $W \oplus X$ (Writable XOR Executable) security model [49, 43], 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.

Return-oriented programming (ROP) attacks [53], bypass the $W \oplus X$ model by using only code already residing in the process's memory space. The adversary combines short instruction sequences from different locations in memory, whereas each sequence ends with a *return* instruction that enables the chained execution of multiple instruction sequences. The ROP attack method has been shown to be Turing-complete and its applicability has been demonstrated on a broad range of architectures: x86 [53], Atmel AVR [24], SPARC [8], ARM [38], Z80 [12], and PowerPC [41].

ROP attacks are increasingly used in practice, in particular, the recent ROP-based attacks on well-established products such as Adobe Reader [36, 50], Adobe Flashplayer [3], or Quicktime Player [28]. Moreover, ROP has been also adapted to kernel exploits: Hund et al. [32] presented a ROP-based rootkit for the Windows operating system which bypasses kernel integrity protection mechanisms. ROP attacks have been also launched on Apple iPhone to perform a jailbreak [30] or to steal a user's SMS database [35]. Finally, tools have been developed enabling the automatic identification of instruction sequences and gadgets [32, 38, 22].

However, ROP can not bypass address space layout randomization (ASLR), a well-known memory protection mechanism available for Linux [49] and Windows [31]. Basically, ASLR randomizes the base addresses of memory and code segments so that the adversary can no longer predict start addresses of instruction sequences. However, recent attacks show that ASLR is often vulnerable to information leakage attacks [57, 52, 60] allowing adversaries to gain useful information on the memory layout of the running process. This in turn allows the adversary to calculate the start addresses of the instruction sequences. Moreover, several ASLR instantiations do not randomize all memory segments of a process, or are unable to randomize several dynamic libraries, be-

ASIACCS '11, March 22–24, 2011, Hong Kong, China.

[1] National Institute of Standards and Technology

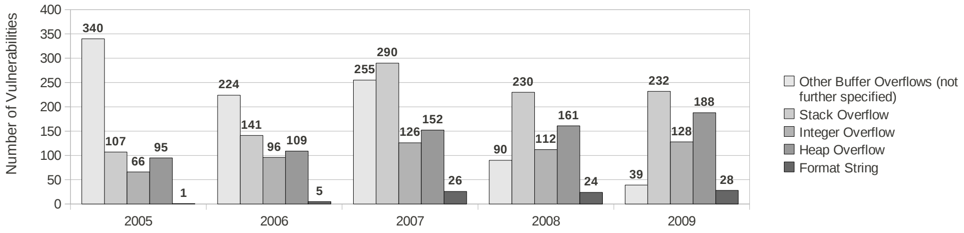


Figure 1: Buffer overflow vulnerabilities from 2005 to 2009

cause these are not ASLR compatible. Hence, the adversary still has a large enough code space to mount a ROP attack as shown in [51, 39].

Existing Tools and Countermeasures.

We already mentioned that ROP bypasses $W \oplus X$ and can be also applied to ALSR protected programs (e.g., [51, 39]). 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, 59, 15, 40, 48]; *instrumentation*-based solutions [37, 16, 1, 2, 29, 55]; and *hardware*-facilitated solutions [26, 25]. However, as we discuss in detail in related work (Section 6), the existing solutions suffer from various shortcomings and practical deficiencies: They either cannot provide *complete detection* of ROP attacks [16, 29, 37], or require *side information* such as debugging information [1, 2] or source code [19, 59, 15, 40, 48], 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 end-user's perspective not always sufficient, because they will only be effective if *all* software vendors really employ 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) allowing ROP attacks.

Our Contributions.

We present the design and implementation of *ROPdefender*, a practical tool that enforces return address protection to detect ROP attacks. We improve existing proposals by detecting unintended return instructions issued in a ROP attack without requiring any side information. 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[2]. However, we developed a new Pintool, *ROPdefender*, that enforces return address checks at runtime. One of our main design goals was to create a practical tool that can be used without the need to change hardware. Hence, we aimed to adopt already existing techniques such as shadow stack [15, 59, 26] for return addresses, and the concept of binary instrumentation as used in taint tracking [47, 17] or return address protection [29, 16, 37]. In particular, our contributions are:

- **Defense technique:** *ROPdefender* detects sophisticated ROP attacks (based on return instructions) without requiring specific side information. As proof of concept we show in Section 5.2 that *ROPdefender* detects recent ROP-based exploits in Adobe Reader.
- **Flexibility and interoperability:** *ROPdefender* can be applied to complex multi-threaded applications such as Adobe Reader or Mozilla Firefox. It can be deployed on Windows and Linux for Intel x86 based systems requiring no new hardware features. As we will discuss in Section 4, *ROPdefender* is able to handle a wide range of exceptions which violate the calling convention.
- **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 and discuss how the performance of *ROPdefender* could be improved.

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 [56]. Lastly, it should be mentioned that our current implementation does not detect the recent ROP attack [11] which uses indirect jumps rather than returns. We will discuss in Section 5.3 how such ROP attacks can be addressed in the future.

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

Return-oriented programming (ROP) is basically a generalization of return-into-libc [56, 44] attacks. It allows an adversary to induce arbitrary program behavior without injecting any malicious code. Rather than returning to functions in libc, ROP returns to short instruction sequences each terminating in a *return* instruction. The return instruction ensures that one sequence is executed after the other: It pops the address of the subsequent instruction sequence from the stack and transfers control to it. Multiple instruction sequences can be combined to a *gadget* which represents a particular atomic task (e.g., load, store, add,

[2] Moreover, it has been used in [61] for a checksum-aware fuzzing tool and in [17] as dynamic taint analysis system.

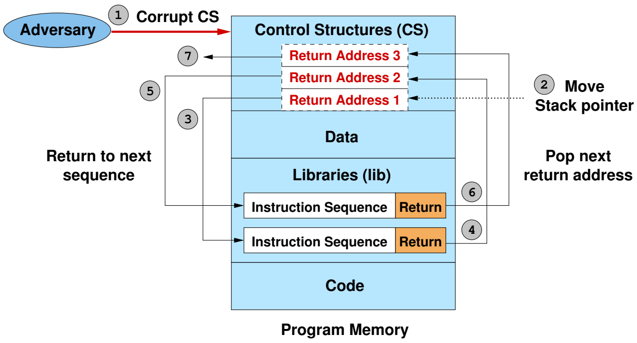


Figure 2: A general ROP attack

etc.). The attack technique is considered Turing-complete, if the adversary is able to construct gadgets for all basic operations: memory operations, arithmetic and logical operations, (un)conditional jumps, and system calls.

2.1 High-Level Idea of ROP

Figure 2 illustrates the general ROP attack. It 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, e.g., a buffer overflow. Hence, the adversary is able to overwrite control-flow information on the CS section (the stack), e.g., the return address of a vulnerable function (step 1). The adversary injects several return addresses each pointing to an instruction sequence in the lib section. In step 2, the stack pointer (SP) is moved to the first return address. If the adversary uses conventional stack smashing techniques [4] (i.e., overwriting the return address of a function), the value of SP will be automatically changed to this position. This is because return address 1 is injected at the place where the original return address was located. Upon function return, execution is not redirected to the original calling function but instead to an instruction sequence in the lib section (step 3). This sequence is terminated by another return instruction which pops return address 2 from the CS section (step 4) and redirects execution to the next instruction sequence (step 5). This procedure is repeated until the adversary terminates the program.

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 [53, 8, 24, 38, 41, 12, 32, 51]. A new ROP attack has been recently presented in [11] 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.

Unintended Instruction Sequences.

On Intel x86, ROP attacks are in particular dangerous due to the presence of the so-called *unintended* instruction sequences. These can be issued by jumping in the middle of a valid instruction resulting in a new instruction sequence never intended by the programmer nor the compiler. These sequences can be easily found on the x86 architecture due to variable-length instructions and unaligned memory access. Consider for instance the following x86 code with the given intended instruction sequence:

```
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, the following unintended instruction sequence would be executed by an Intel x86 processor:

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

In particular, solutions only securing returns in function epilogues are not able to detect ROP attacks that are based on these sequences. We will describe in the next section how to instantiate a ROP attack that is only based on unintended instruction sequences.

2.2 Why Protecting Returns in Function Epilogues Does Not Help

There exists several compiler and instrumentation-based solutions that aim to detect return address attacks [15, 26, 59, 16, 29, 55]. 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, they check if the return address has been modified. We show in the following that countermeasures (integrated in compilers or based on instrumentation techniques) checking only intended returns can not prevent ROP attacks that are based on unintended instruction sequences.

As mentioned in Section 2.1, the first steps of a ROP attack include: (Step 2) moving the stack pointer (SP) to the first return address and (Step 3) redirecting execution to the first instruction sequence, i.e., changing the instruction pointer (IP) to the first instruction of instruction sequence 1. However, return address checkers like [15, 26, 59, 16, 29] can prevent the ROP attack if these two steps are performed by overwriting the return address of a vulnerable function, because these tools perform a return address check in the function epilogue. In order to avoid detection by such countermeasures, these two steps have to performed without using a return instruction in an intended function epilogue. Further, the instruction sequences executed must be unintended to bypass checks for intended return instructions. Unintended sequences can be often found on Intel x86 as shown in [53].[3]

There exists several attack techniques for gaining control over SP (Step 2) and IP (Step 3) without using intended returns. For instance, the 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. Instead of overwriting a return address, the adversary could overwrite *pointers*, e.g., function pointers or entries of the Global Offset Table (GOT)[4]. If an adversary overwrites such a pointer, and the pointer is used as a jump target afterwards, the 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 return address 1. In general, this can be performed by a

[3] The instruction sequence search algorithm (GALILEO) proposed in [53] avoids intended function epilogues.

[4] In Unix-based systems the GOT holds absolute virtual addresses to library functions.

stack-pivot sequence allowing to change SP to an arbitrary value [20]. For instance, on Intel x86 this can be achieved by pointing IP to the following (unintended) sequences:

```
xchg %esp,%eax; ret
mov %ecx, %esp; ret
```

Since the first sequence exchanges (xchg) the contents of %eax and %esp (SP), it requires the %eax register to contain the desired value of SP. The second sequence moves the content of %ecx to %esp. Therefore, the adversay has to load the desired value of SP into %ecx before. The final return instruction of both sequences has the effect that return address 1 is popped from the stack and execution is redirected to instruction sequence 1. Note that both sequences must be unintended to bypass countermeasures that check (intended) returns in function epilogues. Note that further attack techniques on how to instantiate a ROP attack without using a return instruction are also discussed in [11].

3. OUR SOLUTION

In this section we present our security architecture to detect and defeat ROP attacks using return instructions.

3.1 Assumptions

In the following we briefly discuss the main assumptions in our model.

1. **Adversary:** The adversary is able to launch a ROP attack which *cannot* be detected by compiler or instrumentation based solutions that only secure returns in function epilogues (see Section 2.2 for an example).

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

3. **Security measures:** We assume that the hardware and the operating system enforce the $W \oplus X$ security model. Modern processors and operating systems already enable $W \oplus X$ by default.

4. **Trusted Computing Base:** The adversary cannot tamper with 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 the kernel, verification or extensive testing as well as load-time integrity checking of the software components belonging to our tool.

3.2 High-Level Description

Since we assume no access to source code (Assumption 2), we make use of *instrumentation*. Basically, it allows to add extra code to a program to observe and debug the program's behavior [45]. We use a *shadow stack* to store a copy of the return address once a function is called. We instrument all return instructions that are issued during program execution and perform a return address check. Our approach is similar to existing shadow stack approaches, e.g., [15, 59]. However, in contrast to existing approaches, *ROPdefender* (i) checks each return instruction issued to the processor, which detects even unintended instruction sequences, and (ii) it handles various special cases that are necessary for a practical defense tool.

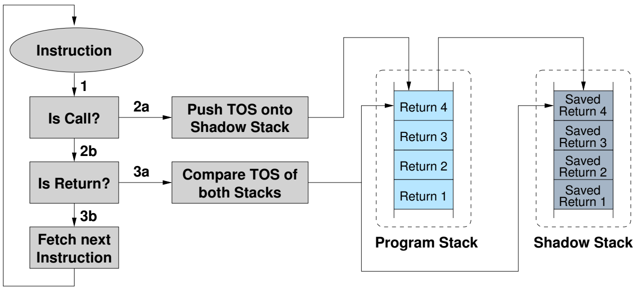


Figure 3: Our high-level approach

Our high-level solution for detecting ROP attacks is depicted in Figure 3. 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 a *shadow stack* (transition 2a in Figure 3).

Otherwise, if the instruction is a return 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 3). 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.

According to the Intel x86 calling convention [33], 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.

3.3 Tools and Techniques

As mentioned above, we use *instrumentation* to detect ROP attacks. Generally, instrumentation can be performed at runtime or at compile-time. For our purpose we focus on dynamic binary instrumentation at runtime to avoid access to side information. Generally, there are two classes of dynamic binary instrumentation frameworks: (i) probe-based and (ii) just-in-time (jit) compiler-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 [46], DynamoRIO [7], and Pin [42] use a just-in-time compiler. Unlike 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: 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 (see Assumption 4). Figure 4 depicts our proposed architecture.

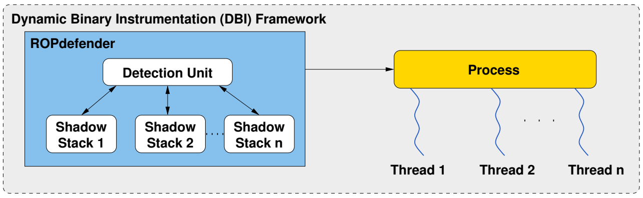


Figure 4: General architecture of *ROPdefender*

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 code which then enforces the return address check.

ROPdefender consists of several shadow stacks and a detection unit. 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 DETAILS

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 motivation 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.[5] Intel uses Pin in the Intel Parallel Studio [34] 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 5 shows the instantiation of our architecture consisting of the Pin Framework and the Pintool *ROPdefender*. 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.

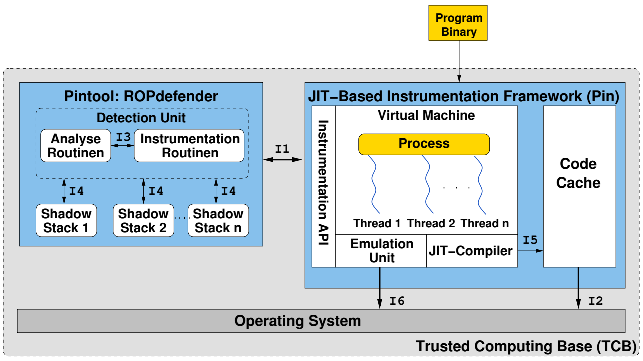


Figure 5: Implementation of *ROPdefender* within the binary instrumentation framework Pin

Pin is configured by Pintools. Basically, Pintools allow to specify 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

[5] Pin has been also used as fuzzing tool [61] and as dynamic taint analysis system [17].

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 3 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 3. 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, 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, 55, 37]), 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 first class consider a chain of various function calls: A calls B, B calls C, and C calls D. According to the calling convention, all functions must return explicitly after completing their task: D returns to C, C to B, and B to A. However, the system calls *setjmp* and *longjmp* allow to bypass multiple stack frames, which means that before the return instruction of D has been reached, execution is redirected back to A, although B, C, and D have not yet returned. Hence, *ROPdefender* expects the return of D, but the program issues the return of A. 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 "misuses" 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 is enforced by a combination of 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[6] 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

[6] 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.

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[7] 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)` after *_Unwind_ RaiseException_Phase2* returns legally.

5. EVALUATION

In this section we evaluate the performance of *ROPdefender*, show how it detects a recent exploit, and finally, we discuss ROP attacks without returns.

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 [58] using the reference inputs. Figure 6(b) and 6(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 [47] (based on Valgrind). According to the results in [17, 47], applications running under these tools are from 30x to 50x times slower which is enormously higher compared to *ROPdefender*. Also DROP [13] causes an average slowdown of 5.3x.

To increase the performance of *ROPdefender*, we can either improve the Pin framework 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.

Our experiments also show that it is possible to apply *ROPdefender* to large applications used in everyday life such as Mozilla Firefox. We were able to browse websites and watch Internet videos with an acceptable time delay without *ROPdefender* raising a false positive. Moreover, we compared *ROPdefender* (running in the user space) to the kernel-level instrumentation tool DTrace [10]. Tracing system calls with DTrace induced such a high overhead that the Mozilla Firefox browser was not usable anymore.

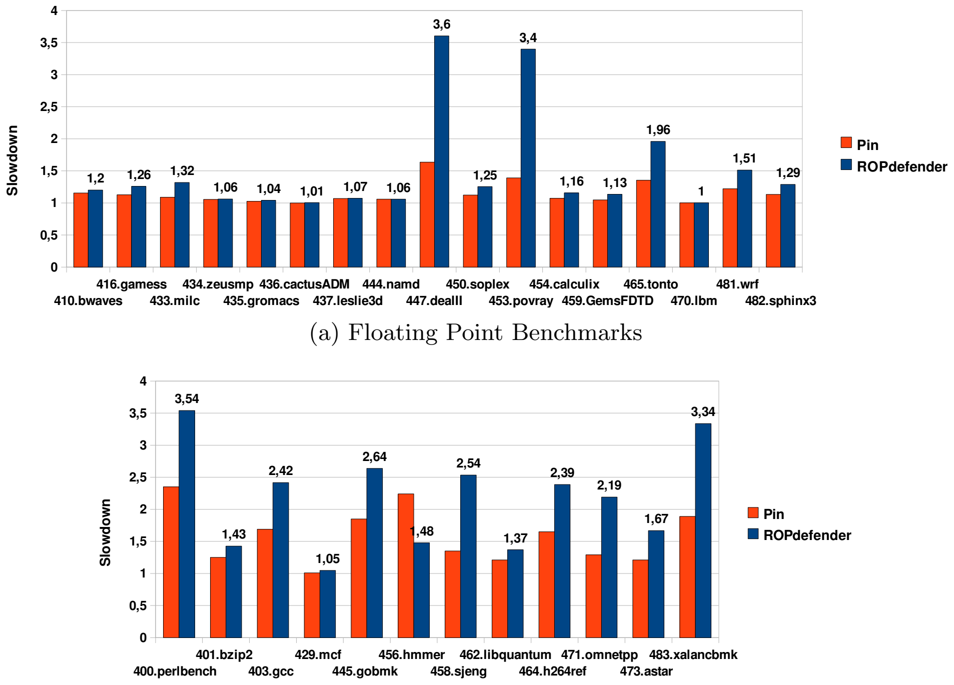
5.2 Case Study

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 [36]. Generally, the attack in [36] 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* successfully detects 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. 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 `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 (see first row in Table 1). To show that *ROPdefender*

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



(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 6: SPEC CPU2006 Benchmark Results

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
BIBLockSmithAssert NoLocksImpl	0x0700A8AC	NULL	0x0700A8B0
...	...	...	...

Table 1: Detection of ROP Attack on Adobe Reader

detects all malicious returns, we temporarily allow the exploit to continue. As can be seen from Table 1, *ROPdefender* also detects the subsequent malicious returns. All following expected return addresses are NULL, because the shadow stack is empty after the first mismatch.

5.3 Discussion

ROPdefender detects all ROP attacks based on returns and any attack that is based on corrupting a return address, e.g., conventional stack smashing [4] or return-into-libc [56]. Recently, Checkoway et al. [11] presented a new ROP attack that is only based on indirect jump instructions rather on returns. Since *ROPdefender* does currently only check return instructions, it can not detect this attack. ROP attacks without returns simulate the return instruction by a pop-jump sequence which pops the address of an instruction sequence from the stack and afterwards redirects control to it by an indirect jump instruction. Since such pop-jump sequences (even not unintended) rarely occur in practice, the attack uses the following technique: Each instruction sequence ends in an indirect jump to a single pop-jump sequence that acts as a trampoline after each instruction sequence. In order to defend against ROP without returns, *ROPdefender* would have to decide at runtime if an indirect jump is a legal one or not. Since there exists no convention regarding the target of an indirect jump (in contrast to returns), it seems impossible to defend against such ROP attacks without having some information about the program's structure. However, indirect jumps do not occur as frequently as returns and the concrete implementation presented in [11] makes 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.

In parallel work, Chen et al. [14] extended *ROPdefender*'s return address protection and showed first techniques to defeat ROP without returns based on jit-instrumentation. The main idea of their approach is that an indirect jump must remain within the boundaries of the function from where it has been issued. Hence, the adversary can no longer enforce a jump from one library function to another one. However, he is still able to jump to an arbitrary instruction within the current function. Moreover, as mentioned in [14], there are a few exceptions where an indirect jump targets an instruction outside of the current function, e.g., tail calls or indirect jumps used in the PLT (Procedure Linkage Table) to redirect execution to the GOT (Global Offset Table). In our future work we aim to address these problems and aim to extend *ROPdefender* allowing it to efficiently detect ROP attacks without returns. In particular, we aim to detect any misuse of an indirect jump instruction to completely prevent ROP without returns. For this we plan to integrate *ROPdefender* into a control-flow integrity framework.

6. RELATED WORK

In the following we explore countermeasures against return address attacks and discuss their shortcomings.

6.1 Randomization

Address Space Layout Randomization (ASLR) [49, 31] 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 mounting brute-force attacks [54] or through information leakage attacks [57, 52, 60] allowing adversaries to gain useful information on the memory layout. 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 [51, 39]. Roglia et al. [51] also propose to encrypt function addresses contained in the Global Offset Table (GOT) to prevent their ROP attack. 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 if adversaries are able to bypass ASLR.

6.2 Compiler-Based Solutions

Below we discuss different compiler-based approaches that aim to mitigate return address attacks. One problem they all have in common is that they require recompilation and access to source code. Further, they are not able to detect ROP attacks based on unintended instruction sequences.

StackGuard [19] places a dummy value, referred to as *canary*, below the return address on the stack. 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 [59] and Return Address Defender (RAD) [15] guard return addresses by holding copies of them in a safe memory area (i.e., on the shadow stack).

Finally, two compiler-based solutions were developed in parallel to our work that specifically address ROP attacks [40, 48]. Li et al. [40] developed a compiler-based solution against return-oriented kernel rootkits [32]. First, all unintended return instructions are eliminated through code transformation. Second, the intended return instructions are protected by a technique referred to as return indirection: Call instructions push a return index onto the stack which points to a return address table entry. The return address table contains valid return addresses the kernel is allowed to use. 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, *ROPdefender* requires no access to source code and also addresses exceptional cases which might occur during ordinary program execution.

A noteworthy compiler-based approach is G-Free [48] that defeats ROP attacks through gadget-less binaries. In contrast to the aforementioned approach and to *ROPdefender*, G-Free also addresses ROP attacks without returns. Basically, G-Free guarantees during compilation that the resulting binary does not contain unintended instruction sequences. Intended return instructions are encrypted against a random cookie created at runtime. Moreover, (intended) indirect jumps (and calls) are only allowed if the function (from where the indirect jump originates) has been entered through a valid entry point. This prevents an adversary from executing indirect jumps outside the currently executing functions. As proof of concept, G-Free has been applied to GNU libc. However, to provide full protection against ROP attacks, each linked library and the original program code have to be compiled with G-Free, which might in practice result in false positives if a library is not compatible to G-Free. Although G-Free already prevents ROP attacks without returns, *ROPdefender* does not need programs and libraries to be recompiled.

6.3 Instrumentation-Based Solutions

Securing Function Epilogues.

There exist two works [29, 16] that aim to detect malicious changes of return addresses by using probe-based instrumentation techniques. Both approaches instrument function prologues and epilogues to incorporate a return address check on each function return. However, as we already described in Section 2.2, both approaches are not able to detect ROP attacks that use unintended instruction sequences because they only instrument *intended* function epilogues.

Control-Flow Integrity.

XFI [2] enforces control-flow integrity (CFI) [1] which guarantees that program execution follows a Control-Flow Graph (CFG) created at load-time. It 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 return address protection. While XFI only instruments intended return instructions, it also checks whether indirect jumps or calls follow a valid path in the CFG. This makes it hard, if not impossible, for an adversary to launch the attack even with unintended instructions. However, XFI mainly suffers from practical deficiencies: The binary instrumentation framework Vulcan [23] used by XFI is not publicly available and is restricted to Windows. More importantly, to build the CFG, XFI requires some information on the program's structure which are extracted from Windows debugging information files (PDB files). These are not provided by default. In contrast, our *ROPdefender* requires no side information and is based on an open source framework.

Measuring Frequency of Returns.

Chen et al. [13] and Davi et al. [21] 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 sequence after, e.g., each second instruction sequence.

Just-in-Time Instrumentation.

Program Shepherding [37] is based on the jit-based instrumentation framework DynamoRIO and monitors control-flow transfers to enforce a security policy. It instruments direct and indirect branch instructions with the goal to prevent execution of malicious code. As part of its restricted control-flow policy it also provides 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 can still return and invoke 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).

TRUSS (Transparent Runtime Shadow Stack) [55] is another tool based on DynamoRIO. 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., [47, 17]) 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, 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 [47] and DYTAN [17]). However, we believe that *ROPdefender* can be incorporated into existing taint analysis systems.

6.4 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 x86 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.

StackGhost [26] is another hardware-facilitated solution, but available on SPARC systems. StackGhost is based on stack cookies that are XORed against return addresses. 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. Note further that 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) is a powerful attack that bypasses current security mechanisms widely used in today's computing platforms. It enables the adversary to perform Turing-complete computation without injecting any new code and executing instruction sequences never intended by the original programmer.

The main contribution of our work is to present a practical countermeasure against ROP attacks (based on return instructions) without requiring access to side information such as source code or debugging information. In this paper, we presented our *ROPdefender* that fulfills accurately this requirement and that is able to detect and 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. Further, we need also protection against return address attacks targeting the operating system that *ROPdefender* relies on. However, we still need to have measures against ROP without returns. Currently, we are working on countermeasures against ROP attacks without returns and on a countermeasure against ROP for embedded systems.

8. ACKNOWLEDGMENTS

We thank Hovav Shacham and Stephen Checkoway for the fruitful discussions on return-oriented programming attacks based on indirect jumps. The first author was supported by EU FP7 project CACE.

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. 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] M. Abadi, M. Budiu, U. Erlingsson, G. C. Necula, and M. 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. USENIX Association, 2006.

[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] D. 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] E. Buchanan, R. Roemer, H. Shacham, and S. 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] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[10] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference*, pages 15–28. USENIX Association, 2004.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–572. ACM, 2010.

[12] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of EVT/WOTE 2009*, 2009.

[13] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In A. Prakash and I. Gupta, editors, *Fifth International Conference on Information Systems Security (ICISS 2010)*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2009.

[14] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient detection of the return-oriented programming malicious code. In *Sixth International Conference on Information Systems Security (ICISS 2010)*, volume 6503 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2010.

[15] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, pages 409–417. IEEE Computer Society, 2001.

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

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

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 91–104. USENIX Association, 2003.

[19] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. 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. USENIX Association, 1998.

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

[21] L. Davi, A.-R. Sadeghi, and M. 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.

[22] T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

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

[24] A. Francillon and C. 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. ACM, 2008.

[25] A. Francillon, D. Perito, and C. 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] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 55–66. USENIX Association, 2001.

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

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

[29] S. Gupta, P. Pratap, H. 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. ACM, 2006.

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

[31] M. Howard and M. Thomlinson. Windows vista isv security. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`, Apr. 2007.

[32] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 2009.

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

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

[35] V. Iozzo and R.-P. 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.

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

[37] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206. USENIX Association, 2002.

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

[39] L. Le. Payload already inside: data re-use for ROP exploits. In *Black Hat USA*, July 2010.

[40] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with ”return-less” kernels. In *Proceedings of the 5th European conference on Computer systems*, EuroSys ’10, pages 195–208. ACM, 2010.

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

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM, June 2005.

[43] Microsoft. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[44] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), 2001.

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

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

[47] J. Newsome and D. 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.

[48] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC’10, Annual Computer Security Applications Conference*, Dec. 2010.

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

[50] S. Ragan. Adobe confirms zero-day - rop used to bypass windows defenses. http://www.thetechherald.com/article.php/201036/6128/, 2010.

[51] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*. IEEE, 2009.

[52] H. 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.

[53] H. 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.

[54] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. 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. ACM, 2004.

[55] S. Sinnadurai, Q. Zhao, and W. 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.

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

[57] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista. http://www.phreedom.org/research/bypassing-browser-memory-protections/, Aug. 2008. Presented at Black Hat 2008.

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

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

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

[61] T. Wang, T. Wei, G. Gu, and W. 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)*. IEEE Computer Society, 2010.