Detecting Stack Layout Corruptions with Robust Stack Unwinding

Yangchun Fu[1,2], Junghwan Rhee[1(✉)], Zhiqiang Lin[2], Zhichun Li[1], Hui Zhang[1], and Guofei Jiang[1]

[1] NEC Laboratories America, Princeton, USA
{rhee,zhichun,huizhang,gfj}@nec-labs.com
[2] University of Texas at Dallas, Richardson, USA
{yangchun.fu,zhiqiang.lin}@utdallas.edu

Abstract. The stack is a critical memory structure to ensure the correct execution of programs because control flow changes through the data stored in it, such as return addresses and function pointers. Thus the stack has been a popular target by many attacks and exploits like stack smashing attacks and return-oriented programming (ROP). We present a novel system to detect the corruption of the stack layout using a robust stack unwinding technique and detailed stack layouts extracted from the stack unwinding information for exception handling widely available in off-the-shelf binaries. Our evaluation with real-world ROP exploits has demonstrated successful detection of them with performance overhead of only 3.93 % on average transparently without accessing any source code or debugging symbols of a protected binary.

Keywords: Stack layout corruption · Stack layout invariants · Stack unwinding information · Return oriented programming

1 Introduction

The stack is a critical memory structure to ensure the correct execution of programs since control flow changes through the values stored in it (e.g., return addresses and function pointers). Therefore, the stack has been a popular target of many attacks and exploits [9,33,36,46,48,51] in the security domain. For instance, the stack smashing attack [33,36,48,51] is a traditional technique that has been used to compromise programs. Recently return oriented programming (ROP) [9,46] has gained significant attention due to its strong capability of compromising vulnerable programs in spite of up-to-date defense mechanisms, such as canaries [17], data execution prevention (DEP) [32], and address space layout randomization (ASLR) [54] under certain conditions (e.g., memory disclosure vulnerabilities, and the low entropy of ASLR).

Such attacks manipulate one aspect of the stack regarding return addresses to hijack execution. However, the stack not only contains return addresses but

Y. Fu—Work done during an internship at NEC Laboratories America, Princeton.

F. Monrose et al. (Eds.): RAID 2016, LNCS 9854, pp. 71–94, 2016.
DOI: 10.1007/978-3-319-45719-2_4

also stores many other data, such as local variables and frame pointers, with specific rules on its layout for a correct execution state. These rules are statically constructed by a compiler precisely for each function. Unfortunately such constraints on the stack layout are not strictly checked by the CPU as evidenced by the aforementioned attacks allowed, but a correct program execution strictly follows such constraints and they are in fact parsed and checked when needed (e.g., exception handling, backtrace in debug). Our intuition is that the current ROP attacks are not aimed to follow these stack layout constraints. Thus the inspection of the stack layout could be an effective inspection method to detect ROP attacks based on the manipulation and the side-effects in the stack layout. Our method is applicable to multiple stack-based attacks that tamper with the stack layout (Sect. 3), but we focus on ROP attacks in this paper since it is one of the most sophisticated and challenging attacks to date.

While many approaches have been proposed to detect and prevent ROP attacks [16,19,38], they are not without limitations. In particular, many of them heavily rely on the patterns of ROP gadgets, e.g., the length of a gadget, and the number of consequent gadgets. As such, attacks violating these patterns keep emerging, as witnessed by the recent attacks [12,25].

An early exploration toward this direction, ROPGuard [21], detects ROP attacks by unwinding stack frames using a heuristic approach, based on the stack frame pointer [3] (i.e., `ebp`-based stack unwinding in Windows). This is one way to check the sanity of the stack with an assumption on the compiler's practice. Unfortunately, its detection policy is not general in many operating systems causing a failure to protect the programs compiled without the stack frame pointers. For instance, from the version 4.6 of GCC (the GNU Compiler Collection), the frame pointer option (`-fomit-frame-pointer`) is omitted by default for the 32-bit Linux making this approach unreliable.

In this paper we present a novel systematic approach called SLIck[1] to verify the stack layout structure at runtime with accurate and detailed information, which is generated by a compiler for exception handling [1,2] and available inside the binaries. From this information, we extract stack layout invariants that must hold at all times. We show verifying these invariants is effective for detecting the stack manipulation caused by ROP attacks overcoming the limitations of previous approaches based on stack unwinding. For our approach to be practical, this information should not be optional during compilation, or require source code since in many environments a program is deployed in the binary format. A pleasant surprise is that the stack frame layout information is widely available in Linux ELF binaries stored in the `.eh_frame` section due to the support of exception handling (even for C code). Moreover, this binary section is required in the `x86_64` application binary interface (ABI) [6].

The contribution of this paper is summarized as follows:

– We present two novel security invariants of the stack regarding *legitimate return address chains* and *legitimate code addresses* based on the data stored

[1] SLIck represents **S**tack **L**ayout **I**nvariants **C**hec**k**er similar to `fsck`.

in the `.eh_frame`. While the `.eh_frame` provides the information regarding the stack layout, it is not directly applicable to ROP detection. The invariants proposed in this paper fill this gap.
- We present a novel ROP detection technique based on stack layout invariants and a robust stack unwinding. This mechanism improves the robustness of a prior heuristic-based stack unwinding (e.g., `ebp`-based [21]), which fails to inspect the binaries that are not compiled with frame pointer support.
- We propose flexible stack unwinding algorithm to overcome a general and practical challenge in stack unwinding approaches which fail to unwind the entire stack due to the incompleteness of stack frame information. Our evaluation shows this instance is quite often, which leads to frequent false negative cases of the stack inspections without addressing this issue.

2 Background

2.1 Return Oriented Programming

Return oriented programming (ROP) is an offensive technique that reuses pieces of existing code chained together to create malicious logic. An attacker identifies a set of instruction sequences called *gadgets* linked together using payloads, which traditionally are placed in the stack [46] transferring control flow via the return instructions. Recently the attack pattern became diverse involving the call or jump instructions, which can trigger an indirect control flow [13] and the payloads can be also placed in other places, such as the heap [49].

2.2 Stack Frame Information in Binaries for Exception Handling

When a program executes, many low level operations occur in the stack. Whenever a function is called, its execution context (e.g., a return address) is pushed to the stack. Also many operations, such as handling local variables, delivering function call parameters, the flush of registers, occur on the stack exactly as they are determined during the compilation time. The specific rules on how to use each byte of each stack frame are predetermined and embedded in the program.

Figure 1 illustrates an example of this stack layout information taken from a function (`ngx_pcalloc`) of `nginx`, a high-performance HTTP server and reverse proxy. The top of the figure shows a part of its disassembled code. The middle part of the figure shows an example of the stack layout information, which is organized with the reference to the head of each stack frame. The memory address of a stack frame is referred to as the Canonical Frame Address (CFA) [1,2], which is the stack pointer address at the function call site.

The decoded information at the bottom illustrates the detailed stack layout at each instruction. For instance, [`40530c: cfa=32(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)`] shows the exact locations of the top of the current stack frame (`cfa=`), the pushed register values (`rbx=, rbp=`), and the return address (`ret=`) described in terms of the stack pointer address and the offsets at

Disassembled code

```
00000000004052fb <ngx_pcalloc>:
 4052fb:   48 89 5c 24 f0          mov    %rbx,-0x10(%rsp)
 405300:   48 89 6c 24 f8          mov    %rbp,-0x8(%rsp)
 405305:   48 83 ec 18             sub    $0x18,%rsp
 405309:   48 89 f5                mov    %rsi,%rbp
 40530c:   e8 2d fe ff ff          callq  40513e <ngx_palloc>
 ...       ...                     ...
 405336:   48 83 c4 18             add    $0x18,%rsp
 40533a:   c3                      retq
```

Stack layout information from .eh_frame

```
00000530 000000000000001c 00000534 FDE cie=00000000
  pc=00000000004052fb..000000000040533b
  DW_CFA_advance_loc: 14 to 0000000000405309
  DW_CFA_def_cfa_offset: 32
  DW_CFA_offset: r6 (rbp) at cfa-16
  DW_CFA_offset: r3 (rbx) at cfa-24
  DW_CFA_advance_loc: 49 to 000000000040533a
  DW_CFA_def_cfa_offset: 8
  ...
```

Decoded stack layout information

```
  4052fb: cfa=08(rsp),                                    ret=-8(cfa)
  405300: cfa=08(rsp), rbx=-24(cfa),                      ret=-8(cfa)
  405305: cfa=08(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)
  405309: cfa=32(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)
  40530c: cfa=32(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)
  ...                                                        ...
  405336: cfa=32(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)
  40533a: cfa=08(rsp), rbx=-24(cfa), rbp=-16(cfa), ret=-8(cfa)
```

Fig. 1. A detailed view of the stack layout information of the Nginx binary.

the instruction at `0x40530c`. This information shows the detailed rules on the stack usage which were not considered by the current ROP attacks to evade.

While we have found that the stack frame information is useful for the detection of ROP exploits, to be a practical solution, this information should be widely available in binaries. Modern programming languages mostly support exception handling. To do so, the runtime environment should be capable of interpreting and unwinding stack frames such that the exception handler can correctly respond to the exceptions. The ELF binary format, which is widely used in the Linux and BSD operating systems, stores it in the `.eh_frame` and `.eh_framehdr` sections [2]. Similar information is also available in other platforms to support exception handling. For instance, the Windows OS has an exception handling mechanism called Structured Exception Handling (SEH) [5,42]. The mach-O [4] binary format used by Apple Macintosh programs has similar binary sections (`.eh_frame`, `.cfi_startproc`, and `.cfi_def_cfa_offset`).

Our investigation shows that the `.eh_frame` section is included by default in the compilation using the `gcc` and `g++` compilers for C and C++ programs. According to the definition of the application binary interface (ABI) for `x86_64`, it is a required section for a binary [6]. The `strip` utility with the most strict option (e.g., `strip --strip-all`) does not affect this section. In addition, most binaries deployed in modern Linux distributions include this section.

For instance, in Ubuntu 12.04 64 bit version all binaries in the `/bin` directory have a valid `.eh_frame` section. Among the entire set of the program binaries examined, over 97 % of around 1700 binaries have this information except special binaries: the Linux kernel image (e.g., `kernel.img`) and the binaries compiled with `klibc`, which is a special minimalistic version of the C library used for producing kernel drivers or the code executed in the early stage of a booting.

3 Overview of SLIck

We use the stack layout information available from the binary section for exception handling to detect ROP exploits. As a research prototype, we present SLIck, a robust stack unwinding based approach that does not rely on any gadget patterns, such as a gadget sequence, or behavior. Previous approaches (e.g., [16,19,38]) are based on the characteristics of ROP gadgets, such as call-precedence or the length of gadget sequences, which make them vulnerable to new attacks [12,25]. The overview of SLIck is illustrated in Fig. 2.

SLIck uses two invariants regarding the stack layout information (to be shortly described in Sect. 4 in details) to detect an ROP attack.

- **Stack Frame Chain Invariant (Sect. 4.1).** The stack frame information inside the binary describes how stack frames must be chained, and the unwinding of the runtime stack information should not be different from it.
- **Stack Frame Local Storage Invariant (Sect. 4.2).** The accumulated stack operations in a function are summarized as a constant because the memory usage in each stack frame should be cleaned up when the function returns.

SLIck inspects the runtime status of the monitored program's stack regarding these two invariants transparently and efficiently so that ROP attacks can be precisely detected. SLIck has two major system components.

- **Derivation of stack layout invariants (Sect. 4).** To achieve efficient runtime checks, the necessary information is derived in an offline binary analysis. Given a binary executable as an input, this component extracts the stack frame information from the `.eh_frame` section and constructs stack layout invariants. Also, the table of valid instructions of this binary is derived to verify the stack frame local storage invariant.

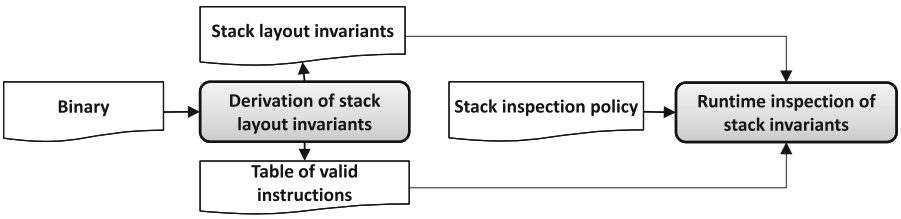

Fig. 2. System overview of SLIck.

– **Runtime inspection of stack invariants (Sect. 5).** This component verifies whether stack invariants hold at runtime and detects any violation caused. SLICK inspects the stack status when an OS event is triggered to avoid high overhead of fine-grained techniques [14,19]. Diverse OS events with different characteristics can be used to trigger the inspection as policies. For instance, the inspection on all system calls will catch the ROP behavior that uses any system services, such as a file access, and network usage. Using the timer interrupts, which trigger the context switches, enables non-deterministic inspection points that make it hard to accurately determine our inspection time and also enables frequent inspections in the CPU intensive workload.

Adversary Model and Assumptions. We consider an adversary who is able to launch a user-level stack-based return oriented programming (ROP) attack, which modifies the stack to inject its payload using a native program in the ELF format with the stack frame layout information widely available in the Linux platform. There is no assumption on the characteristics of gadget content (e.g., a sequence, a length, and the call-precedence of gadgets) which can be used in the attack.

The techniques in this paper are in the context of Linux and the ELF binary format because the mechanism and the implementation of the stack layout information is specific to each OS platform due to the distinct underlying structures of OSes. However, we believe a similar direction can be explored in other OS platforms which are described in the discussion section.

We assume that the integrity of the operating system kernel is not compromised and the ROP attack is not towards the vulnerability and the compromise of the kernel. While such attack scenarios of ROP exploits are realistic, in this paper we do not focus on the countermeasures for such attacks because of the existing detection and prevention mechanisms on OS kernel integrity [22,23,27,39–41,45]. We rely on such approaches to ensure the integrity of OS kernel and SLICK, which is designed to be a module of it.

Finally, we mainly focus on native programs for the detection of ROP exploits. The programs based on dynamically generated code running on virtual machines, interpreters, and dynamic binary translators have their own unique structures on their runtime and the stack layout. Currently, we do not focus on ROP defense for these binaries.

4 Derivation of Stack Layout Invariants

Rich stack layout information of the `.eh_frame` section can be used to derive potentially many invariants regarding the layout of the stack. In this paper, we focus on two invariants that are motivated by the following challenges.

First, ROP attacks can manipulate the valid chains of the function calls of the original program, and determining such manipulation robustly and transparently is a remaining challenge. Recent approaches on control-flow integrity have made substantial progress particularly when they can access or transform

source code [18,56]. Some approaches attempted to achieve a practical control-flow integrity by relaxing strict control-flow [60,61]. However, they still introduce new attacks [24]. Second, ROP gadgets popularly utilize unintended instructions and it is non-trivial to detect such usage efficiently. We introduce two stack layout invariants to solve these challenges.

4.1 Stack Frame Chain Invariant (FCI)

Observation. The description regarding the head of a stack frame (CFA), can validate how far a previous stack frame should be apart from the current one. For instance, the information [`40530c: cfa=32(rsp), ..., ret=-8(cfa)`] in Fig. 1 shows that the CFA is at the address stored in the *rsp* register plus 32, and the return address is at $ret = -8(cfa)$ which is resolved as $rsp + 32 - 8$ using the location of the CFA. This information enables the validation of the linkage of stack frames.

Invariant. For an instruction c in a function, let us define the accumulation of stack operations between the function prologue and c in terms of a stack distance as $BL(c)$ (Backward stack frame Layout). This information generated by a compiler for the instruction c is retrieved from the CFA of `.eh_frame`. For instance, the return address at $B6$ in Fig. 3, $BL(B6)$ is $-12(SP)$ (i.e., stack pointer + 12 bytes) due to three decrements of the stack pointer (each by 4 bytes) for local variables. A runtime version, $BL'(c)$, is subject to manipulation under attacks requiring the verification whether it conforms to $BL(c)$ for all stack frames in a chain. This invariant is presented as $BL(c) = BL'(c)$ called the Stack Frame Chain Invariant (FCI).

Verification. SLIck checks this invariant using a stack unwinding algorithm (Sect. 5) iteratively over all stack frames validating the integrity of the BLs as a chain. Any inconsistency in one of the BLs in the chain causes cascading effects in the following stack frames, therefore, breaking the BL sequence in the unwinding

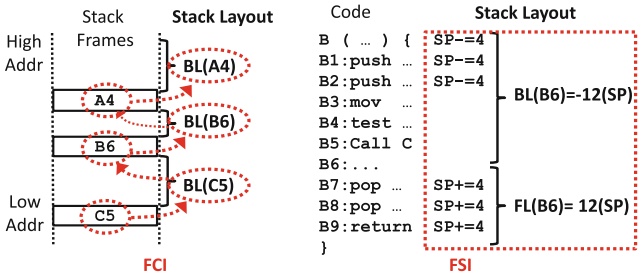

Fig. 3. Illustration of stack layout invariants.

procedure. SLICK determines this invariant is satisfied if the unwinding procedure over all stack frames is successful. To perform this runtime verification efficiently, we precompute the BLs using the CFAs from the `.eh_frame` section.

4.2 Stack Frame Local Storage Invariant (FSI)

Observation. Programs use the stack to store data (e.g., for local variables and register spills). To limit the impact across stack frames, the allocation and deallocation of local stack memory in a frame should be *paired up* so that the stack memory usage for a function could be cleaned up when the function returns.

Invariant. This observation regarding the gross sum of local stack operations is summarized as follows. Let us define the accumulated stack operations between the code c and the function epilogue in terms of a stack distance as $FL(c)$ (Forward stack frame Layout). The observation on the stack local storage is represented as $BL(c)+FL(c) = k$, which we call the Stack Frame Local Storage Invariant (FSI). In the right figure of Fig. 3, $BL(B6)$ is $-12(SP)$, and $FL(B6)$ is $12(SP)$ leading to $k = 0$. Typically k should be zero except the special corner cases where functions do not properly return such as the `exit`. This invariant allows to determine the usage of unintended code popularly used in ROP attacks because such code may not follow the original code's semantic.

Verification. To efficiently check whether the executed code conforms to this invariant, we precompute a table of instructions originally intended in the program, named as a *table of valid code addresses (TVC)*. Its rows show all possible code addresses (i.e., every byte offset of the code including unintended code in the program) and the column indicates a boolean state whether the code is valid (T) or invalid (F) depending on the $BL(c) + FL(c)$.

We use the `.eh_frame` and a binary analysis for the computation of this table. The instructions derived from the stack frame information are marked as valid. However, due to its compressed structure, which mainly describes the instructions involving stack operations, not every instruction is covered. For such cases, we use a binary analysis to simulate the instructions and determine the validity. SLICK applies this check as part of a stack unwinding algorithm.

SLICK considers that a program is compromised if either or both of these two invariants are violated. We present more specific details on how to check them at runtime in Sect. 5.

5 Runtime Inspection of Stack Invariants

In this section, we present how SLICK inspects stack invariants and robustly detects their violations.

5.1 Practical Challenges

After we use a traditional stack unwinding algorithm [3] to inspect the invariants, we have identified the cases that frustrate the current algorithm and limit the inspection of the full stack. There are two cases categorized.

Failure type	Description	Attributes of virtual memory pages		Binary exist exist	Unwind info
		Type	Page permission		
Type A	Incomplete unwinding info	Code	Executable	Yes	No
Type B	Invalid unwinding	Not found	Not executable	*	*

Type A: Incomplete Unwinding Information. We found that a rare portion of code in terms of coverage has incomplete unwinding information mainly in the low level libraries and the starting point of a program. It is important to address this issue because such code stays in the stack during execution and there is a high chance to face it during the unwinding. If this issue happens, the vanilla stack unwinding algorithm cannot proceed the unwinding procedure due to the missing location of the next stack frame.

Based on our experiments over 34 programs including widely used server applications and benchmark, the cases that we identified are summarized into mainly three cases. First, it is triggered by the entry point of `ld`, which is the dynamic linker and loader in Linux. Second, the first stack frame which is the start of the program can generate a type A error. The third case is the `init` section of the `pthread` library.

Type B: Invalid Unwinding Status. Unlike type A, this case should not occur in benign execution. However, this incorrect execution state is observed when the stack layout is manipulated by attacks. The stack unwinding algorithm strictly verifies the validity of the stack layout information formulated by the compiler across all stack frames. Any single discrepancy due to stack manipulation leads to invalid unwinding conditions. Specifically this case is characterized as the state shown in the table: the return code address obtained from the stack is not found from the executable memory area.

Type A failures can block the full inspection of all stack frames in stack unwinding-based approaches. Therefore, this issue must be addressed to achieve robust stack unwinding. We address it using *flexible stack unwinding*, which is a novel variant of the stack unwinding algorithm that enables robust detection of type B errors while addressing type A errors. Next, we present the details of our algorithm that inspects stack invariants based on the flexible stack unwinding.

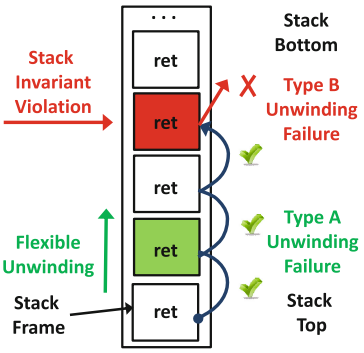

Fig. 4. Flexible unwinding.

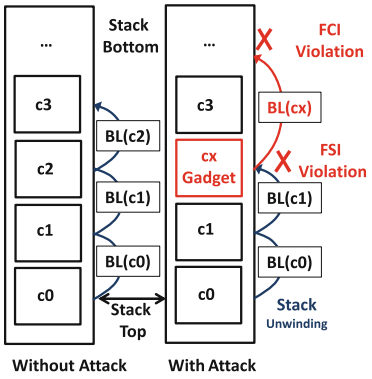

Fig. 5. Stack invariant violation.

5.2 Stack Invariant Inspection Algorithm

Figures 4 and 5 presents a high level illustration of our algorithm to inspect stack invariants while addressing the practical stack unwinding challenges. When the inspection is triggered, Algorithm 1 inspects stack frames starting from the top to the bottom of the stack as shown in Fig. 4. The algorithm bypasses type A failures while detecting type B errors caused by the violation of stack invariants illustrated in Fig. 5. FCI and FSI violations are respectively caused by an illegitimate chain of stack frames and return addresses.

The Algorithm 1 is triggered by an operating system event (e.g., a system call, an interrupt) represented as the `OsEvent` function (Line 1). This function executes the `SIInspect` function (Line 2), the main logic for stack inspection. When this function detects either a type B error (FCI violation) or an invalid return code instruction (FSI violation) during stack unwinding, it returns `Fail`. Upon the detection of any violation, the `PostProcess` function is called (Line 4) to stop the current process and store the current context for a forensic analysis.

Unwinding Library Code. A program executes the code for multiple libraries as well as the main binary. Such libraries have separate stack frame information in their binaries which are loaded into distinct virtual memory areas (`VMA`). During the scan of the stack, our algorithm dynamically switches the VMA structure for the return address (Line 11), which is implemented as two nested loops in `SIInspect`; the outer `while` loop (Lines 10–38) switches different VMAs while the return addresses in the same binary are efficiently handled by the inner loop (Lines 17–27) without searching for another VMA.

For each VMA, the `.eh_frame` information is retrieved from the binary (Line 16). For each code address, the algorithm checks its validity (Line 20). If it is valid and the return code stays in the same VMA, the `GetNextRet` function is called to unwind one stack frame. Otherwise the algorithm returns a violation of FSI at Line 21. This loop is repeated to unwind following stack frames as

Algorithm 1. Stack Invariant Inspection Algorithm

```
    SZ = sizeof(UNSIGNED LONG)
 1: function OSEvent(REGS)
 2:     Ret = SIInspect(UserStack, REGS)
 3:     if Ret == Fail then
 4:         PostProcess(REGS, UserStack)
 5:     return
 6: function SIInspect(UserStack, REGS)
 7:     CFA = REGS → SP; VMA = GetVMA(CFA); UnwindDepth = 0
 8:     StackTop = REGS → SP; StackBot = GetStackStart(VMA, CFA)
 9:     InvalidInstr = False
10:     while true do                                              ▷ Outer loop
11:         VMA = GetVMA(REGS → IP)
12:         if VMA is invalid or not executable then
13:             return Fail                                        ▷ Type B, FCI violation
14:         if VMA → VM_FILE does not exist then                   ▷ Dynamic code
15:             Goto DoFlexibleSIInspect
16:         EH = GetEHSection(VMA)
17:         do                                                     ▷ Inner loop
18:             if REGS → SP < StackTop or REGS → SP >= StackBot then
19:                 return Fail                                    ▷ Type B, FCI violation, Stack Pivot Detection
20:             if TVC [REGS → IP] == False then                   ▷ FSI violation
21:                 return Fail
22:             if REGS → IP > VMA → VM_Start and REGS → IP < VMA → VM_End then
23:                 UnwindDepth += 1
24:             else
25:                 Ret = GotNext; break                           ▷ Find another VMA
26:             Ret = GetNextRET(CFA, REGS, EH, StackBot, StackTop)
27:         while Ret == GotNext
28:         if Ret is NoUnwindingInfo then                         ▷ Type A
29: :DoFlexibleSIInspect
30:             offset = FlexibleSIInspect(REGS → SP, StackBot)
31:             if offset is EndOfStack then
32:                 return Success
33:             else
34:                 REGS → SP += offset; REGS → IP = *(REGS → SP) of UserStack,
35:                 REGS → BP = *(REGS → SP - SZ) of UserStack; REGS → SP += SZ
36:                 CFA = REGS → SP
37:         else if Ret is Invalid then
38:             return Fail                                        ▷ Type B, FCI violation
39:     return Success
40: function FlexibleSIInspect(Start_SP, StackBot)
41:     for SP = Start_SP; SP < StackBot; SP += SZ do
42:         IP = *SP; VMA = GetVMA(IP)
43:         if VMA is valid and VMA → VM_FILE is available then
44:             return SP - Start_SP
45:     return EndOfStack
```

long as the function returns `GotNext`. For code c, its $BL(c)$ is returned by the `GetNextRet` function. If a return address is replaced by k, a manipulated value, the divergence $BL(k) = BL'(c) \neq BL(c)$ will cause cascading effects on unwinding of the following stack frames. Any mismatch of a single stack frame with its unwinding information causes a violation of FCI at Line 13, 19, or 38.

Stack Pivot Detection. During stack unwinding, Algorithm 1 performs various checks to ensure precise unwinding and detect anomaly cases. A popular technique in recent ROP attacks is stack pivoting [49,62] that changes the location of stack to the manipulated content (e.g., heap). This attack is trivially detected by our algorithm (Line 18) because SLIck can distinguish an invalid stack memory address.

Flexible Stack Unwinding. To handle type A failures, we provide *flexible stack unwinding* algorithm (Lines 40–45). When a type A case happens, the `FlexibleSIInspect` function advances the stack pointer in a brute force way and checks whether a legitimate stack frame appears next. If the return address found in this search belongs to a code section based on its memory address range and the corresponding file, this function returns the offset of the stack. And then the algorithm goes back to the outer loop (Line 10), and the stack layout information of the new stack frame is examined. If it is a type B case, the `GetNextRET` function will return `Invalid` in the next loop. If it turns out to be a type A case again, it will go back to the `FlexibleSIInspect` function by returning `NoUnwindingInfo`. Lastly, if it is a valid frame, it will be unwound and takes a following loop iteration.

5.3 Stack Inspection Policies

SLICK inspects the runtime status of a program stack based on the policies regarding which types of OS events trigger the inspection. Here we present two policies used for our evaluation (Sect. 6.3) and our framework allows user defined policies as well.

System Call Inspection (SYS). This policy checks the stack on all system calls which provide lower level services to the program, such as memory allocation, file operations, network operations, and a change of memory permission. They are the typical targets of ROP exploits to achieve functionality beyond the original program, and this policy provides a cost-effective inspection at the intermediary points of OS operations to observe high impact system activities.

System Call and Non-deterministic Inspection (SYS+INT). This policy achieves finer-grained inspection by narrowing down the gaps between the inspections and making inspection intervals non-deterministic by using non-deterministic OS events, such as interrupts. As an attack scenario against SLICK, an ROP exploit may attempt to predict SLICK's inspection time and clean up the stack manipulation to hide its evidence. Since this scheme uses non-deterministic OS events to perform inspections, this attack becomes significantly hard to be successful. This scheme can be further strengthened by increasing the randomness, e.g., by performing additional inspections with random intervals.

6 Evaluation

In this section, we present the evaluation of SLICK in the following perspectives.

- How effective is SLICK at detecting real-world ROP exploits?
- What is the impact of SLICK on benign programs?
- How efficient is SLICK for inspecting stack invariants?

We have implemented SLICK for 32 bit and 64 bit Ubuntu 12.04 LTS Linux systems as a kernel module and user level tools for offline analyses.

6.1 Detection of ROP Attacks

We applied SLICK on 7 real-world ROP exploits available in Linux of 32 bit and 64 bit architectures. Table 1 presents the details of the program's runtime status and the detection results by SLICK.

Table 1. Detection of stack invariant violations of ROP exploits. The number of unwinding failures (#F-unwind) is generally correlated with the number of events (# Events), but it can be higher if multiple stack frames have failures.

Program			Syscall inspection policy			Invariant violation		Attack description	
Name	Ver	Env	#Events	$\|\|S\|\|$	#F-unwind	Detection	Type	Exploit info	Syscall
Nginx	1.4	64 bit	100452	22	96895	✓	FSI	CVE-2013-2028	sys_write
Mysql	5.0.45	64 bit	2128	13	2156	✓	FCI & FSI	CVE-2008-0226	sys_execve
Nginx	1.4	32 bit	42937	22	40231	✓	FCI & FSI	CVE-2013-2028	sys_write
Mysql	5.0.45	32 bit	2027	12	1792	✓	FCI	CVE-2008-0226	sys_rt_sigaction
Unrar	4.0	32 bit	141	10	142	✓	FCI	CVE-2007-0855	sys_write
HT Editor	2.0.20	32 bit	292	13	326	✓	FCI	CVE-2012-5867	sys_lstat64
MiniUPnPd	1.0	32 bit	56	8	50	✓	FCI	CVE-2013-0230	sys_time

The first three columns show the description of the program, its name (Name), version (Ver.), and the architecture that it runs on (Env). For this experiment, we use the system call inspection policy. The 4th, 5th, and 6th columns show the runtime status: the number of system call events (# Events), the average stack depth ($||S||$) during the execution, and the number of type A stack unwinding errors (# F-unwind) that flexible stack unwinding algorithm successfully addressed. The next two columns show the detection of ROP exploits based on stack invariant inspection: the "Detection" column shows whether the violation of an invariant is detected. Our algorithm stops a program on the first violation of an invariant which could be either an FCI or an FSI. If both of them occur in the same iteration of algorithm, it is presented as FCI & FSI. The type of violation is presented in the "Type" column. Exploit information (Exploit Info) and the system call at the time of detection (Syscall) are presented in the next columns.

We experimented with real-world exploits against widely used server and desktop software: Nginx, Mysql, Unrar, HT Editor, and MiniUPnPd. These software and the ROP exploits have different characteristics shown as various numbers of system calls and the depths of the stack. All tested ROP exploits are successfully detected due to violations of stack invariants.

6.2 Impact on Benign Programs

For a practical usage of SLICK, it should have low false positives in benign programs. For this evaluation, we used total 34 programs from popular open source projects and benchmarks: 3 widely used server programs (Nginx, Apache, Mysql), a CPU benchmark for Linux (NBench), a data compression utility (7zip), and 29 programs from the SPEC 2006 benchmark. The stack invariants are inspected with two inspection policies: the system call inspection policy (SYS), and the system call and non-deterministic inspection policy (SYS+INT).

Table 2 summarizes our results. The first column describes the program name. We present the data for two inspection policies in different groups of columns. The next three columns describe the evaluation using the SYS inspection policy. The following three columns show the result using the SYS+INT policy. The SYS+INT inspection policy increases the number of inspection events in the CPU intensive benchmarks more significantly (e.g., 434.zeusmp has over 28 times higher events because of timer interrupts). I/O intensive programs get most timer interrupts from the kernel code, such as another interrupts or system calls. Such cases are not additionally inspected because the programs are already checked on the transition from the user mode to the kernel mode. This policy can harden the inspection of CPU intensive programs that have a low number of system calls. Timer interrupts capture a program call stack at arbitrary non-deterministic execution points. Therefore, the average call stack depth ($||S||$) is different between two experiments in many cases of the SPEC benchmark.

In general the number of type A failures that flexible unwinding addresses (#F-unwind) is highly correlated with the number of inspection events (# Events). One reason for this behavior is that the first stack frame created on the start of the program stays in the stack and triggers a type A failure on each system call. Another reason is the pthread library; large programs using multiple threads get additional type A errors due to this library.

While most of programs triggered non-trivial number of type A failures, in all cases, no violation of stack invariants is detected causing zero false positives of our approach.

6.3 Performance Analysis

We evaluate the runtime performance impact of SLICK on the protected programs in the prior evaluation. The overhead is related to the frequency of the inspections and the depth of stack unwinding. SLICK is configured to scan the full stack. We used the apache bench with the load of a thousand requests to generate the workload for Apache and Nginx webservers. The performance of the Mysql database and the 7zip tool are measured using the packaged benchmarking suites. The Nbench and SPEC 2006 benchmarks are executed using the standard setting. Performance numbers from different types of benchmarks are normalized in a relative way so that the performance of native execution becomes 1. Our measurement data are presented in Fig. 6. We present SLICK's performance in two inspection policies.

Table 2. Stack invariant inspection of benign applications. No violation is detected.

Program name	SYS			SYS+INT		
	#Events	$\|\|S\|\|$	#F-unwind	#Events	$\|\|S\|\|$	#F-unwind
Nginx	16164	16	16171	16183	16	16190
Apache	24466	15	24472	24481	15	24488
Mysql	40347778	12	40377139	40451780	12	40481318
Nbench	87371	7	87371	163973	7	163973
7zip	59922	8	74874	68516	8	82650
400.perlbench	35361	12	35361	35666	12	35666
401.bzip	450	7	450	4649	8	4649
403.gcc	714	15	714	1578	13	1568
410.bwaves	993	9	993	9048	9	9048
416.gamess	16848	17	16848	17057	17	17057
429.mcf	1023	8	1023	2945	7	2945
433.milc	19092	11	19092	26020	10	26020
434.zeusmp	258	8	258	7422	6	7422
435.gromacs	3009	15	3009	3651	14	3651
436.cactusADM	2115	16	2115	3869	14	3869
437.leslie3d	303	9	303	7646	5	7646
444.namd	6159	9	6159	14018	7	14018
445.gobmk	8799	12	8799	21492	23	21492
450.soplex	360	9	360	372	9	372
453.povray	5040	21	5040	5386	21	5386
454.calculix	537	9	537	568	9	568
456.hmmer	207	9	207	2168	5	2168
458.sjeng	1671	12	1671	4511	15	4511
459.GemsFDTD	1626	9	1626	2825	7	2825
462.libquantum	120	8	120	149	8	149
464.h264ref	1236	8	1236	9909	11	9909
465.tonto	6303	16	6303	6743	16	6743
470.lbm	2091	8	2091	3280	6	3280
471.omnetpp	570	12	570	829	11	829
473.astar	783	9	783	7371	6	7371
481.wrf	5598	17	5598	7666	15	7666
482.sphinx3	9213	12	9231	10510	11	10510
988.specrand	378	10	378	384	9	384
999.sperand	378	10	378	385	9	385

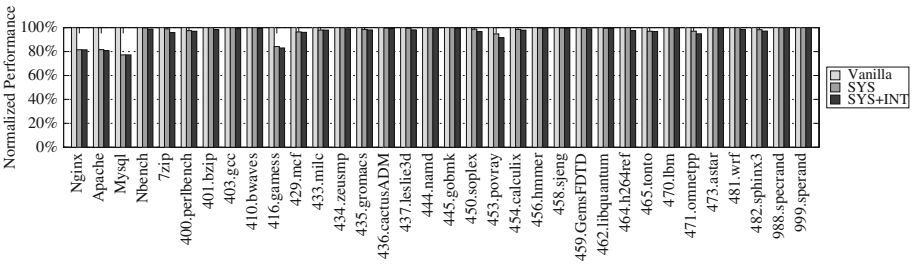

Fig. 6. Runtime performance of SLICK.

Runtime Impact for the SYS Policy. With this policy, the average overhead of SLICK in 34 evaluated programs is 3.09 % with the maximum overhead of 22.8 % in Mysql. High overhead of Mysql is due to very intensive stress tests for a database in I/O and low level services. As an evidence, the second column of Table 2 which is the system call count shows that Mysql benchmark generates a significantly higher number of system calls over 40 millions compared to other programs having under 164 thousands system calls.

Runtime Impact for the SYS+INT Policy. A finer-grained inspection based on system calls and timer interrupts is offered with a slightly higher performance cost of 3.93 % on average and with the maximum overhead of 22.8 % in Mysql. This policy causes increased overhead on CPU intensive workloads due to additional inspections introduced by timer interrupts.

7 Discussion

Comparison with ROPGuard/EMET. Our work is closely related to an early exploration based on stack unwinding, ROPGUARD/EMET [21]. However, there are several major differences that distinguish SLICK from ROPGUARD/EMET as follows.

- Our approach has a higher and reliable inspection coverage compared to ROPGUARD: SLICK inspects the user stack with the full depth on all system calls and timer interrupts. In contrast, ROPGUARD inspects the stack only on a selective set of critical user level APIs with a limited depth of the stack. Thus ROPGUARD/EMET cannot detect the attacks that directly trigger system calls without using the APIs. Also if the binary is built statically, ROPGUARD cannot be applied due to its base on the library interposition technique.
- ROPGUARD operates in the same user level as the monitored program. Therefore, it is subject to manipulation by the attacker. However, SLICK is isolated from the user space due to its implementation in the OS kernel.
- SLICK's inspections are performed with a higher frequency compared to ROPGUARD: SLICK inspects the stack on all system calls and non-deterministic

timer interrupts. ROPGuard, however, only checks the stack on a set of critical API functions.
- SLICK achieves a reliable stack walking and improved ROP detection by using precise stack layout information extracted from the unwinding data while ROPGuard uses a heuristic based on the frame pointer which is not reliable.
- This work proposes two stack layout invariants which are derived from the unwinding information, and these invariants are verified by using an improved and reliable stack walking mechanism.
- We discovered that a small number of binaries have incomplete unwinding information which affects the current stack walking mechanisms. We propose flexible stack unwinding algorithm to overcome this issue and enable a reliable and high quality inspection of the entire stack.

Stack Pivoting Attacks. A stack pivoting attack [49,62] manipulates the stack pointer to point to the data controlled by ROP gadgets. This attack is trivially detected by our approach because SLICK uses the valid stack memory address ranges assigned by the OS. When an unexpected memory area is used for the stack pointer, SLICK detects it as a FCI violation (Sect. 5.2). A related work [43] achieves this feature using a compiler approach and source code while SLICK can prevent this attack transparently for existing binaries.

Stack Manipulation Detection. Traditional buffer overflow attacks [36,48,51] attempting to overwrite the return addresses in the stack are likely to violate the invariants. Thus such attacks can be transparently detected by SLICK without any recompilation, or instrumentation of the program.

Support of Dynamic Code. Dynamically generated code is used by several platforms such as virtual machines, interpreters, dynamic binary translators, and emulators. While they are out of the scope of this paper, we believe our approach can be extended to support them with engineering efforts because these platforms also provide ways to unwind stack frames for dynamic code. For instance, Java has a tool called `jstack` that can dump the whole stack frames including both of the Java code and the underlying native code and libraries. If such a platform-specific logic is integrated into our stack unwinding algorithm, we should be able to support such dynamic code as well as native code.

Integrity of a Kernel Monitor. SLICK resides in the OS kernel. While the attack scenarios of ROP exploits against the kernel is certainly possible, in this paper we do not focus on the countermeasures for such attacks because of existing detection or prevention mechanisms on OS kernel integrity [22,23,27,39–41,45]. We rely on such approaches to ensure the integrity of SLICK.

Control-Flow Integrity. Our approach raises the bar for ROP exploits by introducing new security invariants on the layout of the stack. Essentially code

and the stack status have correspondence generated by compilers, but it is not strictly enforced at runtime. SLICK verifies this loose correspondence by using a novel variant of a stack unwinding [3] inspecting stack layout invariants.

Control-flow integrity (CFI) [8] provides strong measures to defeat ROP attacks by strictly checking the control-flow of programs. Recent approaches made a significant progress in the compiler-based techniques [18,56] and achieved practical solutions by relaxing strict control-flows [60,61]. When only program binaries are available, the stack frame chain invariant of SLICK provides a practical and transparent alternative to verify the backward chain of control flow while providing a performance benefit and high applicability without requiring source code, program transformation, or a complete control flow.

User-Space-Only Self-hiding ROP Attacks. ROP gadgets are typically used to achieve a new logic which may not exist in the original program. Most real exploits typically make use of the OS level services [57], such as allocating memory and changing its permissions. Technically it is possible to execute ROP gadgets and recover the manipulation of the stack before the transition to the OS to hide the evidence from SLICK. Such user-space-only ROP attacks in practice would be non-trivial to keep track of the manipulated states and implement a clean up logic without stack pivoting which SLICK detects. The inspections on system calls will capture any such attempt on system related activities. Non-deterministic OS events, such as timer interrupts (varying between 4–20 ms in our experiments), and the inspection events with random intervals will make it further difficult for the exploit to precisely predict the inspection time. This advanced attack to hide itself is an aspect that needs further study which is our future work.

Integrity of Stack Frame Information. For a robust detection of ROP attacks, SLICK ensures the integrity of stack frame information as follows. The integrity of this information inside a binary is verified using a file integrity checker [30]. Given the file integrity, SLICK makes its own shadow copy of the `.eh_frame` section copied directly from the binary to prevent any manipulation. However, its copy loaded into the program's memory for exception handling is subject to potential attacks [34]. The OS kernel makes it read-only, but it is not immutable. Thus SLICK enforces the read-only permission on the program's copy to prevent the attack [34].

Attacks Using Binaries Without Stack Frame Information. Our study presented in Sect. 2 shows that most Linux ELF binaries except special binaries, such as a Linux kernel image and kernel drivers, have stack frame information. A typical compilation of programs includes stack frame information to support exceptions and debugging by default. Binaries without stack frame information are not supported by SLICK due to the lack of required information for stack walking. Such unusual binaries can be prevented from running using system wide

program execution policies. For instance, SLIck can prevent the execution of such binaries when stack frame information is lacking.

Implementing SLIck on Other Platforms. This work focuses on the stack layout information in the ELF binary format, which is popular in Linux and BSD environments. However, other OS environments have similar information for exception handling and debugging. For instance, Windows has the `RtlVirtualUnwind` API that can unwind the stack by using the unwind descriptors of the structured exception handling (SEH) tables in the program images, which can be dumped using the `dumpbin` utility with the `/UNWINDINFO` option [7]. Mac OS' main binary format, Mach-O [4], has similar binary sections, such as `.eh_frame`, `.cfi_startproc`, `.cfi_def_cfa_offset` etc. These information can be used to implement a similar function as SLIck in those OS platforms.

Attacks Using Type A Cases for ROP Gadgets. Based on our study of diverse binaries, a very small number of common libraries have the missing unwinding information in a rare portion of their code: the entry point of the dynamic linker and loader, the first stack frame which is the start of the program, and the `init` section of the `pthread` library. Although the portion of code is small and its capability could be limited, it is possible for the attacker to use this code for gadgets. While we have not presented a specific mechanism to defeat this attack, it can be easily prevented by supplementing the incomplete unwinding information because the scope of such code is very limited. Similar to the technique that we used for constructing the TVC, the unwinding information can be generated by emulating the stack operations of the binary code.

8 Related Work

ROP and Related Attacks. Return oriented programming (ROP) [46] is an offensive technique that reuses pieces of existing program code to compromise a vulnerable program and bypass modern security mechanisms, such as DEP and some ASLR implementations [9] under certain conditions (e.g., memory disclosure vulnerabilities or low entropy ASLRs). It has also been applied in other attack vectors, such as rootkits [15,29]. In addition to the local application of this technique, Bittau et al. proposed the blind ROP (BROP) [9] which can remotely find ROP gadgets to fetch the vulnerable binary over network.

Similar to ROP, another type of control-flow transfer attack based on gadget-reuse is jump oriented programming (JOP) [10], which uses jumps instead of returns. Bosman et al. [11] proposed another type of ROP based on the signal handling function which is universal in UNIX systems. This technique called SigReturn Oriented Programming (SROP) is triggered by the manipulated signal frames stored on a user stack.

ROP Defense. Several mitigation techniques were proposed to defend against ROP attacks such as ASLR [26,28,37,44,53,59,60], compiler techniques [31,35], runtime instrumentation techniques [14,19], and hardware techniques [16,38].

ASLR has been used to block code reuse attacks by dynamically assigning the memory addresses of the code and data sections such that the predetermined memory addresses can be illegal. However, in practice some code may not be compatible with this scheme, thereby leaving attack vectors. Also several approaches have shown that it is possible to bypass this scheme based on information leakage or brute-force attacks [20,47,49,50,52,58].

When source code is available, it is possible to remove attack gadgets through a compiler transformation as shown in [31,35]. If source code is not available, dynamic binary instrumentation can be used to monitor the execution and detect ROP attacks. DROP [14] used the length of gadgets and the contiguous length of gadget chains to characterize and detect ROP attacks. ROPDEFENDER [19] uses binary instrumentation to manage a shadow stack which is not tampered by stack manipulation. These approaches in general have a low runtime efficiency due to a high cost of dynamic binary translation.

ROPECKER [16] and KBOUNCER [38] proposed to utilize the Last Branch Record (LBR) registers to efficiently inspect the runtime history. These approaches are established on the assumptions of gadget patterns, such as the short length of gadgets and a long sequence of consecutive gadgets. Unfortunately, new ROP attack techniques showed such gadget-pattern based schemes can be bypassed [12,25]. ROPGUARD [21] (later integrated into the Microsoft EMET [55]) performs stack inspections for a limited depth at selective critical Windows APIs. This inspection unwinds the user stack using the heuristic on the frame pointer which would be limited based on the build conditions; unless programs are compiled to use the frame pointers, they could not be reliably inspected. In contrast, the stack frame information in the `.eh_frame` enables a precise and reliable unwinding regardless of the requirement of the frame pointer.

The comparison between SLICK and related work in Table 3 highlights that SLICK does not have assumptions on the characterization of ROP gadgets. Hence it is not affected by recent attacks [12,25]. Also its stack unwinding technique

Table 3. Comparison of ROP detection approaches. CL: without using a chain length, GL: without using a gadget length. SC: without using source code. RW: without rewriting. RE: Runtime efficiency. RU: Reliable unwinding.

ROP detector	CL	GL	SC	RW	RE	RU	Main techniques
RETURNLESS [31]	✓	✓	✗	✗	✓	-	Gadget removal based on a compiler technique
GFREE [35]	✓	✓	✗	✗	✓	-	Gadget removal based on a compiler technique
DROP [14]	✗	✗	✓	✗	✗	-	ROP detection based on gadget characteristics
ROPDEFENDER [19]	✓	✓	✓	✗	✗	-	Shadow stack and dynamic instrumentation
KBOUNCER [38]	✗	✗	✓	✗	✓	-	Last branch recording and gadget characteristics
ROPECKER [16]	✗	✗	✓	✓	✓	-	Last branch recording and sliding window
ROPGUARD [21]	✓	✓	✓	✗	✓	✗	Stack unwinding based on the frame pointer
SLICK	✓	✓	✓	✓	✓	✓	Stack invariants verified by a reliable stack unwinding

is more reliable based on the precise stack frame information widely available in the binaries of mainstream Linux distributions. These unique properties enable SLICK to achieve a practical solution which does not require source code, or rewriting of the program binaries for ROP detection.

9 Conclusion

We have presented SLICK, a robust and practical detection mechanism of ROP exploits that is not affected by recent attacks based on the violation of previous assumptions on gadget patterns [12,25]. SLICK detects ROP exploits by using stack layout invariants derived from the stack unwinding information for exception handling widely available in Linux binaries. Our evaluation on real-world ROP exploits shows robust and effective detection without any requirements on source code or recompilation while it incurs low overhead.

Acknowledgments. We would like to thank our shepherd, Michalis Polychronakis, and the anonymous reviewers for their insightful comments and feedback. Yangchun Fu and Zhiqiang Lin were supported in part by the AFOSR grant no. FA9550-14-1-0173 and the NSF award no. 1453011. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of any organization.

References

1. Dwarf debugging information format, version 4. http://www.dwarfstd.org/doc/DWARF4.pdf
2. Exception frames. https://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html
3. Exceptions and stack unwinding in C++. http://msdn.microsoft.com/en-us/library/hh254939.aspx
4. Mach-o executables, issue 6 build tools. http://www.objc.io/issue-6/mach-o-executables.html
5. Structured exception handling. http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx
6. System V Application Binary Interface (ABI), AMD64 Architecture Processor Supplement, Draft Version 0.98
7. x64 manual stack reconstruction and stack walking. https://blogs.msdn.microsoft.com/ntdebugging/2010/05/12/x64-manual-stack-reconstruction-and-stack-walking/
8. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of CCS (2005)
9. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D.: Hacking blind. In: Proceedings of IEEE Security and Privacy (2014)
10. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of ASIACCS (2011)
11. Bosman, E., Bos, H.: Framing signals - a return to portable shellcode. In: Proceedings of IEEE Security and Privacy (2014)

12. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: Proceedings of USENIX Security (2014)
13. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of CCS (2010)
14. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: detecting return-oriented programming malicious code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009)
15. Chen, P., Xing, X., Mao, B., Xie, L.: Return-oriented rootkit without returns (on the x86). In: Proceedings of ICICS (2010)
16. Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.H.: ROPecker: a generic and practical approach for defending against ROP attacks. In: Proceedings of NDSS (2014)
17. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of USENIX Security (1998)
18. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: Proceedings of the IEEE Security and Privacy (2014)
19. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of ASIACCS (2011)
20. Durden, T.: Bypassing PaX ASLR protection. Phrack Mag. **59**(9), June 2002. http://www.phrack.org/phrack/59/p59-0x09
21. Fratric, I.: ROPGuard: runtime prevention of return-oriented programming attacks. https://code.google.com/p/ropguard/
22. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. In: Proceedings of SOSP (2003)
23. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of NDSS (2003)
24. Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: Proceedings of IEEE Security and Privacy (2014)
25. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In: Proceedings of USENIX Security (2014)
26. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: where'd my gadgets go? In: Proceedings of IEEE Security and Privacy (2012)
27. Hofmann, O.S., Dunn, A.M., Kim, S., Roy, I., Witchel, E.: Ensuring operating system kernel integrity with OSck. In: Proceedings of ASPLOS (2011)
28. Howard, M., Thomlinson, M.: Windows ISV software security defenses. http://msdn.microsoft.com/en-us/library/bb430720.aspx
29. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proceedings of USENIX Security (2009)
30. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: a file system integrity checker. In: Proceedings of CCS (1994)
31. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: Proceedings of EuroSys (2010)
32. Microsoft: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2 (2008). http://support.microsoft.com/kb/875352
33. Mudge: How to Write Buffer Overflows (1997). http://l0pht.com/advisories/bufero.html
34. Oakley, J., Bratus, S.: Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. In: Proceedings of WOOT (2011)

35. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proceedings of ACSAC (2010)
36. Aleph One: Smashing the stack for fun and profit. Phrack 7(49), November 1996. http://www.phrack.com/issues.html?issue=49&id=14
37. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In: Proceedings of IEEE Security and Privacy (2012)
38. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Proceedings of USENIX Security (2013)
39. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: Proceedings of USENIX Security (2004)
40. Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proceedings of USENIX Security (2006)
41. Petroni Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of CCS (2007)
42. Pietrek, M.: A crash course on the depths of win32 structured exception handling. Microsoft Syst. J. **12**(1), January 1997
43. Prakash, A., Yin, H.: Defeating ROP through denial of stack pivot. In: ACSAC (2015)
44. Roglia, G.F., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: Proceedings of ACSAC (2009)
45. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of SOSP (2007)
46. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of CCS (2007)
47. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of CCS (2004)
48. Smith, N.P.: Stack Smashing Vulnerabilities in the UNIX Operating System (2000)
49. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Proceedings of IEEE Security and Privacy (2013)
50. Sotirov, A., Dowd, M.: Bypassing browser memory protections in windows vista. http://www.phreedom.org/research/bypassing-browser-memory-protections/
51. Spafford, E.H.: The internet worm program: an analysis. SIGCOMM Comput. Commun. Rev. **19**, 17–57 (1989)
52. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: Proceedings of EuroSec (2009)
53. PaX Team: http://pax.grsecurity.net/
54. PaX Team: Pax address space layout randomization (ASLR) (2003). http://pax.grsecurity.net/docs/aslr.txt
55. The Enhanced Mitigation Experience Toolkit, Microsoft. http://technet.microsoft.com/en-us/security/
56. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Proceedings of USENIX Security (2014)
57. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Proceedings of RAID (2011)
58. Vreugdenhil, P.: Pwn2own 2010: Windows 7 internet explorer 8 exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf

59. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of CCS (2012)
60. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: Proceedings of IEEE Security and Privacy (2013)
61. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Proceedings of the USENIX Security (2013)
62. Zovi, D.A.D.: Return oriented exploitation. In: Blackhat (2010)