A Principled Approach for ROP Defense

Rui Qiao
Stony Brook University
ruqiao@cs.stonybrook.edu

Mingwei Zhang[*]
Intel Labs
mingwei.zhang@intel.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

ABSTRACT

Return-Oriented Programming (ROP) is an effective attack technique that can escape modern defenses such as DEP. ROP is based on repeated abuse of existing code snippets ending with return instructions (called gadgets), as compared to using injected code. Several defense mechanisms have been proposed to counter ROP by enforcing policies on the targets of return instructions, and/or their frequency. However, these policies have been repeatedly bypassed by more advanced ROP attacks. While stricter policies have the potential to thwart ROP, they lead to incompatibilities which discourage their deployment. In this work, we address this challenge by presenting a principled approach for ROP defense. Our experimental evaluation shows that our approach enforces a strong policy, while offering better compatibility and good performance.

1. Introduction

Programs written in C/C++ are not memory safe. Vulnerabilities such as buffer overflow, heap overflow and use-after-free can be exploited by attackers to execute code of their choice. Traditionally, attackers inject payload (called shellcode) into the address space of a victim process, and redirect control to this code. However, with widespread deployment of Data Execution Prevention (DEP), injected code is no longer executable, so attackers have come to rely on code reuse attacks. Return-Oriented Programming (ROP), which chains together a sequence of "gadgets" (code sequences ending with return instructions), is the most powerful and versatile among code reuse attacks. Its power stems from the pervasiveness of returns in binary code. As a result, there are sufficient gadgets in a reasonably large binary to perform Turing-complete computation. Although variants such as Jump-oriented programming have been proposed, ROP remains by far the most dominant code reuse attack, and the only kind used repeatedly in real-world attacks. For this reason, this paper focuses on ROP attacks, and develops a principled approach for defeating them.

Code reuse attacks rely on repeated subversion of control-flows in the victim program. Given the nature of instruction sets on modern processors, such subversion is possible only with indirect control flow transfer instructions. Control-flow integrity (CFI) is a general technique for limiting control-flow subversions by limiting the targets of such indirect control transfers. The idea of CFI is to restrict these targets so that a program's execution follows a control flow graph (CFG) that is computed by a static analysis.

CFI defeats most ROP attacks since they tend to violate the statically computed CFG. However, determined attackers can overcome CFI [21, 16] — specifically, coarse-grained CFI that is based on simple static analyses can be defeated. In fact, researchers have shown that a Turing-complete set of gadgets is available on sufficiently large applications even when coarse-grained CFI is enforced [16].

Recognizing the weakness of coarse-grained CFI against powerful adversaries, researchers have begun to develop techniques for refining CFI policies. These techniques can be broadly classified into those for narrowing down the target of *forward edges,* which include indirect calls and indirect jumps, and those for narrowing down the *backward edges,* consisting of returns. Several forward edge techniques have been developed recently [28, 50, 43, 56, 20], and some of them are already being deployed in production compilers such as gcc and llvm [50]. This factor can greatly reduce the threat of code reuse attacks based on repeated subversion of forward edges, e.g., call-oriented programming [6, 45]. However, mitigating the subversion of backward edges remains a challenge. Although techniques for constraining returns have been known for well over a decade [8, 9], they have not seen wide deployment due to compatibility and performance concerns. It is this factor that motivates our work.

The essential characteristic of ROP is the repeated use of return instructions. Thus, techniques for constraining returns can be very effective in defeating ROP attacks. The primary approach for confining returns is the *shadow stack,* which relies on a second stack that maintains a duplicate copy of every return address. Each call instruction is modified so that it stores a second copy of the return address on the shadow stack. Before each return, the return address on the top of the stack is compared with that atop the shadow stack. A mismatch is indicative of an attack, and program execution can be aborted before a successful control-flow hijack. However, previous shadow stack solutions suffer from one or more of the following drawbacks:

- *Incompleteness.* Many shadow stack schemes are based on compilers [8, 13]. They do not protect returns in hand-written assembly code from low-level libraries such as `glibc` and `ld.so` that are invariably present in every application. Also left unprotected are third-party libraries made available only in binary code form. Moreover, unintended return instructions could be used in ROP, and these won't be checked against the shadow stack.

- *Incompatibility.* In most complex applications, returns don't always match calls. If these exceptional cases are not correctly handled, they lead to false positives that make it difficult (if not impossible) to deploy shadow stack approaches. As a result, experimental evaluation of shadow stack has been performed primarily in the context of sim-

[*]This work was completed when he was at Stony Brook University

[†]This work was supported in part by grants from NSF (CNS-0831298 and CNS-1319137) and ONR (N00014-15-1-2378).

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3682-6/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818021

pler applications such as SPEC benchmarks.

- *High overheads.* Many approaches, especially the more complete approaches that operate on binaries, often suffer from high overheads [17, 11].
- *Lack of systematic protection from all ROP attacks.* None of the previous approaches provide a systematic analysis of possible hijacks of returns, and how these attempts are thwarted. Indeed, most previous approaches incorporate exceptions to the shadow stack policy in order to achieve compatibility. A resourceful adversary can exploit these policy exceptions to carry out successful ROP attacks.

In this paper, we develop a new defense against ROP that overcomes these drawbacks. We provide an overview of our approach below, and summarize our key contributions.

1.1 Approach Overview and Contributions

Our approach is based on the following simple policy:

*Return instructions should transfer control to **intended** return targets.*

If *intention* is interpreted statically, then many existing coarse-grained CFI schemes can be seen as enforcing this policy. However, as discussed before, a static interpretation affords far too many choices for return targets, allowing successful ROP attacks to be mounted. We therefore take a dynamic interpretation of intent. Specifically:

- The ability to return to a location is interpreted as a *one-time use capability.* These capabilities are inferred from and associated with specific parts of the program text, e.g., a call instruction, or, a move instruction that stores a function pointer on the stack, with the intent of using this pointer as the target of a return. A *return capability* is issued each time this program text is executed.
- These return capabilities must be used in a last-in-first-out (LIFO) order. As the term "capability" suggests, not every intended return needs to be taken. Unexercised returns arise naturally due to exception unwinding, thread exits, and so on. However, we require that those return capabilities that are exercised must follow a LIFO order.

The LIFO property of return capabilities means that they can be maintained on a stack, which we will refer to as the return capability stack (RCAP-stack). We make the following contributions in this paper:

- *Strict enforcement:* No exception to the above-described return policy is permitted. This provides a stronger and more principled defense against hijacks of returns, and more generally, of ROP.
- *Static analysis for inferring intended returns:* Strict enforcement implies that all intended returns need to be recorded explicitly on RCAP-stack. While it is easy to infer the intended returns of call instructions, there exist many instances of non-standard returns that don't correspond to any calls. Inferring the target of such non-standard returns poses a challenge. We tackle this challenge by developing a static analysis for this purpose. This analysis has been sufficient to handle almost all non-standard returns we have encountered in our experimental evaluation. Nevertheless, we do incorporate an option for manually annotating code snippets that create return capabilities. This provides a reliable fallback mechanism in cases where static analysis proves insufficient.
- *Better compatibility with existing binaries:* Despite stricter enforcement than previous approaches, our static analysis leads to better compatibility results than previous approaches. Ours is a shadow stack solution that has been shown to be compatible with large and complex COTS binaries, including those containing low-level code and hand-written assembly, *with minimal manual effort.*

2. Background and Threat Model

2.1 ROP Attacks

Return-Oriented Programming (ROP) [46] is the major form of code reuse attacks. ROP makes use of existing code snippets ending with return instructions, which are called gadgets. A gadget used in an ROP attack could either be an intended code sequence, or unaligned code, which refers to unintended instruction sequence beginning from the middle of an intended instruction. This is possible in variable length instruction architectures such as x86. ROP has a property that no matter whether the gadgets are intended or not, in order to perform malicious computation, the program intended control flow has to be changed. We develop an effective policy to break this condition and thus prevent ROP attacks.

2.2 Control-Flow Integrity (CFI)

Control-flow integrity is a security policy that ensures program intended control flows are not subverted. CFI is an important primitive for *secure* static instrumentation [2, 59, 58, 55, 44], as it can limit control flows so that instrumentation cannot be bypassed. Specifically, control flows cannot go to (a) middle of instructions, or (b) an instruction immediately following an added instrumentation that is intended to be executed before that instruction.

Recent works have shown that CFI schemes can be bypassed. Many advanced ROP attacks [21, 16, 5] focus on techniques that can escape control flow policies on backward edges. We want to limit advanced ROP attacks by removing this choice. There also exist advanced code reuse attacks that rely only on forward edges [45, 10, 19], These require techniques complementary to ours that focus on refining of forward edge policies, such as [50], or memory safety solutions.

2.3 Threat Model

We assume a powerful remote attacker that can exploit memory vulnerabilities to read or write arbitrary memory locations, subject to OS-level permission settings on memory pages. We assume the attacker has no local program execution privilege or physical access to the victim system. We also assume the end goal of this attack is to achieve arbitrary code execution, which is a necessary step for further compromises such as installing malware, establishing a persistant base, or escalating privileges.

We assume DEP is enabled on the victim system and therefore ROP is a necessity for payload construction. We assume ASLR is also deployed, but attackers can use memory corruption vulnerabilities to leak the information needed to bypass it without resorting to brute-force.

3. Inferring Intended Control Flow

The first task in enforcing a stronger policy on returns is to precisely infer program intended control flow. This precision is critical because enforcement effectiveness will be directly based on this information.

As discussed earlier, our focus is exclusively on return instructions. We do not attempt to further improve the policies enforced on the remaining types of branch instructions beyond what is already done in our implementation platform, which enforces a coarse-grained CFI property described in [59].

We developed static analysis techniques to identify instructions that push intended return addresses onto stack. As a fallback option, intentions could be captured using manual annotation, but this is minimally needed.

Based on the results of static analysis and/or annotations, instrumentation is added to update RCAP-stack to keep track of the return capabilities acquired by the program by virtue of executing these instructions. RCAP-stack serves as a secure reference database for enforcement.

3.1 Calls

Most call instructions are used for subroutine invocations and therefore express an intention to return to the next instruction. However, it is up to the callee to decide whether the return is actually exercised. For example, a call to `exit()` will never return. Moreover, the call instructions themselves may be used for purposes other than calling functions. For instance, position-independent code (PIC) on x86 uses call instructions to get the current program counter, from which the base of the static data section can be computed in a manner independent of where a certain module is loaded in memory.

Unintended calls do not lead to compatibility problems since we do not require all return capabilities to be used. However, issuing unneeded capabilities can increase an attacker's options. To avoid this, we use a static analysis of target code to determine if the return address generated by a call is definitely discarded before being used by a return instructions. In the simplest case, a discard will happen through the use of a `pop` instruction that pops off the return address at the top of the stack. More generally, the location containing the return address may be overwritten, or the stack pointer incremented to a value greater than this location. If one of these properties holds on every execution path starting at the target of the call, then we conclude that the return address will not be used as the target of a return.

After identifying unintended calls, the remaining calls are instrumented for storing the return address on RCAP-stack. For unintended calls, the RCAP-stack is left unchanged.

3.2 Returns

Returns express an intent to exercise a return capability. We check RCAP-stack if the program has this capability. By default, the return capability is expected to have originated in a previous call, but there are cases where this does not hold, and the return addresses are pushed by other means. Considering that returns using non-call generated return addresses deviate from the standard behavior of returning to callers, we call them *non-standard returns*.

One example of non-standard return is shown in Figure 1. The return instruction (at `0x146bb`) uses a return address generated by a `mov` instruction (at `0x146b4`) rather than a call. This code snippet is taken from GNU's dynamic loader, and the non-standard return is used for dynamic function dispatch after resolving a symbol. Specifically, when a function from another module is called for the very first time, its execution traps to the dynamic loader for symbol resolution. After the loader has resolved the address for the function and cached the result in original module's Global Offset Table (GOT), control should be directed to the called function. This is achieved by first moving the function address (stored in `eax` register) to the top of stack using a `mov` and then issuing a return, as shown in Figure 1.

```
0x146b4 movl %eax,(%esp)
0x146b7 ...
0x146bb ret $0xc
```

Figure 1: A non-standard return from `ld.so`

Note that this non-standard return essentially resembles an indirect jump. However, it has the benefits that (1) no register is clobbered, as compared to a register based indirect jump, and (2) the memory allocated for storing the target of the indirect jump is automatically reclaimed right after, as compared to a memory based indirect jump.

Actually, as will be shown in the next sections, there are a number of such non-standard returns, scattered in different modules. In addition, as compared with standard returns where return addresses saved by calls are used, the intended control flows exercised by non-standard returns are not explicit. Therefore, we need a mechanism for identifying non-standard returns and inferring the intended control flows.

3.3 Illustration of Static Analysis with Real-World Examples

The distinction between a standard and non-standard return is the return address being used. We observe that return address used by a *standard return* is pushed by the call instruction in its caller, and not modified in the callee. On the other hand, return address used by a *non-standard return* is written to the return address stack slot by a non-call instruction in the callee, no matter where it originates. Based on this observation, we have developed a static analysis to identify non-standard returns.

From a high level, our static analysis consists of four steps as discussed below.

Candidate Snippet Extraction. The first step is to extract code snippets that end with returns. After a binary module is disassembled, we build a control flow graph. Then a backward scan is performed on the CFG starting from each return instruction, and up to N instructions (currently set to 30 in our implementation). These snippets are our candidates for analysis.

Semantic Analysis. The second step is to analyze the semantics of each snippet by symbolically executing it. At the beginning of each snippet, each register is assigned a corresponding initial symbolic value. The program state is updated based on the semantics of each executed instruction. At the end of each instruction, each register and modified memory location is associated with expressions of the initial symbols, we call them *semantic equations*. Therefore, the semantic equations after the last instruction (i.e., the return instruction) capture the effects of executing the snippet. Note that the expressions in the equations are normalized by performing variable ordering and constant folding. Therefore, equivalent expressions can be recognized.

Non-standard Return Identification. The next step is to identify non-standard returns. After semantic analysis, the value of stack pointer register before the return instruction can be determined by an expression. Since it is the pointer for the return address slot, if there is any memory write to that location, a non-standard return is identified.

Intended Control Flow Inference. The last step of the analysis is to infer the intended control flow for the non-standard return. To that end, we need to first identify the non-call instruction that stores the return address. We call such an instruction *RAstore*. The semantic equations for each instruction are checked in reverse order to examine writes to the return address slot expression. The first matched instruction is the RAstore, and the source register operand of this instruction captures the intended control flow target.

In the following section, we will describe real world non-standard return examples identified by our analysis.

3.3.1 Non-standard return examples

Previous shadow stack solutions rely on manual identification and ad-hoc instrumentation to support non-standard returns [17, 58, 13]. However, that approach is not scalable. Moreover, it would lead to compatibility issues when applied to large and complex software, making deployment difficult, if not impossible. In this section, we describe several examples of real world non-standard return examples identified by our static analysis. While some of these have been discussed in previous works, the rest have not been recognized or discussed.

Figure 2 shows the non-standard returns in the analyzed code snippets, and analysis results, i.e., semantic equations at return instruction. In the semantic equations, upper case register names (e.g., EAX) are the initial symbolic values, while lower case ones (e.g., eax) indicate the values at end of the analysis. For easier illustration, each code snippet is simplified to only include the last basic block. The semantic equations omit the effects to floating point registers and segment registers. Note that our analysis results do not change when the full code snippets are used and when effects to non-general purpose registers are captured.

The first example is the same one as shown in Figure 1. Analysis indicates that return address comes from `eax`. The analysis discovers the highlighted instruction as the one that pushes the return address.

The second example comes from `setcontext(3)` function of glibc. The single argument of `setcontext` is a pointer to `ucontext_t` structure, which is loaded to `eax` at the first instruction. Since the user context structure contains all saved register information, most of the snippet code performs the job of register restores. Particularly, the program counter placed at offset `0x4c` of `ucontext_t` was loaded to `ecx` at loction `0x3fa81`. And the push instruction at `0x3fa87` pushes it as return address onto stack, which is consumed by the return instruction at the end of the snippet. This non-standard return and the RAstore at `0x3fa87` are identified by our static analysis. Another similar case in function `swapcontext(3)` from the same module, was also identified (not shown in figure).

The third example is a snippet from one of the stack unwinding functions in libgcc_s.so.1. The code first stores `edi`, the address of landing pad (handler code) which is previously computed, to the return address slot of next frame (`0x154cd`). Therefore, the following return will redirect control to the landing pad. This example also demonstrates the power of the analysis: the store to `0x4(%ebp,%esi,1)` at `0x154cd` does not "look" like a return address overwrite, however our static analysis is able to detect it. This is also an example why simple pattern matching based non-standard return identification would not work well.

Code Snippet	Semantics Equations

;; #1 /lib/ld-2.15.so
0x146b0 popl %edx
0x146b1 movl (%esp),%ecx
0x146b4 movl %eax,(%esp)
0x146b7 movl 0x4(%esp),%eax
0x146bb ret $0xc
	eax = *(ESP+8)
	edx = *ESP
	ecx = *(ESP+4)
	esp = ESP + 4
	*(ESP+4) = EAX
	ra = *esp
	= *(ESP+4) = EAX

;; #2 /lib/i386-linux-gnu/libc.so.6
0x3fa73 movl 0x4(%esp),%eax
0x3fa77 movl 0x60(%eax),%ecx
0x3fa7a fldenvl (%ecx)
0x3fa7c movl 0x18(%eax),%ecx
0x3fa7f movl %ecx,%fs
0x3fa81 movl 0x4c(%eax),%ecx
0x3fa84 movl 0x30(%eax),%esp
0x3fa87 pushl %ecx
0x3fa88 movl 0x24(%eax),%edi
0x3fa8b movl 0x28(%eax),%esi
0x3fa8e movl 0x2c(%eax),%ebp
0x3fa91 movl 0x34(%eax),%ebx
0x3fa94 movl 0x38(%eax),%edx
0x3fa97 movl 0x3c(%eax),%ecx
0x3fa9a movl 0x40(%eax),%eax
0x3fa9d ret
	
	eax = *(*(ESP+4)+64)
	edx = *(*(ESP+4)+56)
	ecx = *(*(ESP+4)+60)
	ebx = *(*(ESP+4)+52)
	esi = *(*(ESP+4)+40)
	edi = *(*(ESP+4)+36)
	ebp = *(*(ESP+4)+44)
	esp = *(*(ESP+4)+48)-4
	((*(ESP+4)+48)-4)
	= *(*(ESP+4)+76)
	ra = *esp
	= *(*(*(ESP+4)+48)-4)
	= *(*(ESP+4)+76)

;; #3 /lib/i386-linux-gnu/libgcc_s.so.1
0x154cb movl %esi,%ecx
0x154cd movl %edi,
0x4(%ebp,%esi,1)
0x154d1 addl $0x10,%esp
0x154d4 leal 0x4(%ebp,%ecx,1),%ecx
0x154d8 movl -0x14(%ebp),%eax
0x154db movl -0x10(%ebp),%edx
0x154de movl -0xc(%ebp),%ebx
0x154e1 movl -0x8(%ebp),%esi
0x154e4 movl -0x4(%ebp),%edi
0x154e7 movl 0x0(%ebp),%ebp
0x154ea movl %ecx,%esp
0x154ec ret
	
	eax = *(EBP-20)
	
	edx = *(EBP-16)
	ecx = ESI+EBP+4
	edx = *(EBP-12)
	esi = *(EBP-8)
	edi = *(EBP-4)
	ebp = *EBP
	esp = ESI+EBP+4
	*(ESI+EBP+4) = EDI
	ra = *esp
	= *(ESI+EBP+4)
	= EDI

;; #4 /usr/lib/libunwind-setjmp.so
0x674 pushl %eax
0x675 movl %edx,%eax
0x677 ret
	eax = EDX
	esp = ESP-4
	*(ESP-4) = EAX
	ra = *esp
	= *(ESP-4) = EAX

Figure 2: Code snippets and their analysis results

Our last example is from an unwinding library libunwind. The snippet is simple, and similar to the first example, but used for implementing longjmp.

We note that although the non-standard return compatibility problem has been recognized by many in the literature [17, 49, 13], only the first and third of these four examples have seen manual handling [17]. In contrast, our static analysis systematically identifies all of them, and serves as a basis for automatic instrumentation.

3.4 Discussion

Since that our static analysis is local, it can fail to identify non-standard returns when the RAstore instruction is far away from the return. The only such case we have seen in practice is `makecontext(3)`, which prepares context (stack including return address) for a function to be executed when the context is activated. However, we note that this is a very unique case for supporting special functionalities. In typical scenarios where each execution unit worries about its own context, as stack operations are frequent, for code readability and maintainability reasons the store and the use of return addresses should not be far apart. Even if there exist other such non-local return address modifications, we can fall back to an annotation-based approach. Such annotation may be generated manually or by a compiler. If a compiler generates code that contain RAstore instructions and non-standard returns, they could be annotated. On the other hand, if these special instructions are used by programmers in inline assembly, manual annotation could be performed. In the case of `makecontext`, we used manual annotation.

We note that the annotation approach would be particularly useful in the scenario of a Just-in-Time (JIT) compiler. Since JITted code is generated dynamically, they are not amenable to static analysis. However, non-standard returns do exist in JITted code. For example, deoptimization could use on-stack replacement (OSR) mechanism to replace a stack frame belonging to an inline-optimized function with multiple stack frames, which involves stores of extra return addresses. If the JIT compiler or programmer annotates this, (dynamic) instrumentation will be able to use this information for strict enforcement of intended control flows, similar to what we have done statically.

4. Enforcing Intended Control Flow

In this section, we first describe our intended control flow enforcement based on instrumentation. Then we introduce our technique for supporting RCAP-stack when program has multiple stacks, offering better compatibility. Finally, RCAP-stack protection, the key for enforcement security, is discussed.

4.1 Instrumentation-based Enforcement

Intended control flow enforcement is realized through instrumentation. Specifically, calls, RAstores and returns are involved.

All call instructions, except those identified by target code analysis that have no intention of return, are instrumented. The instrumentation simply pushes the return address saved by call instruction onto RCAP-stack.

Return capabilities issued by RAstores need to be pushed to RCAP-stack too. These could be determined by the static analysis discussed in Section 3.3, Instrumentation is performed on RAstore instruction so that the corresponding return capability is pushed to RCAP-stack.

To completely mediate backward edge control transfers, all return instructions are instrumented to check if required return capability is present on RCAP-stack. Note that due to normal program behaviors such as stack unwinding, the required return capability is not necessarily located at the top of RCAP-stack. Therefore, capability pop continues until a match, similar to previous shadow stack approaches. If such capability is never found when RCAP-stack is empty, this indicates an unintended control flow and suggests an attack. Therefore, an alarm is triggered and the program aborts.

4.2 Transparent RCAP-stack Switching

A program can use multiple stacks, either because it maintains multiple contexts to switch between, or it has multiple threads running concurrently. In these scenarios, if only one RCAP-stack is used, the return capabilities from different contexts could be mixed up, and inappropriate popping of capabilities belonging to other contexts could lead to future false alarms. Therefore, it is necessary to maintain multiple RCAP-stacks.

RCAP-stack needs to be switched once program stack switches. Although intercepting all context switch and thread functions can be performed, it requires significant manual effort. We therefore propose a transparent scheme that uniformly deals with stack switching, no matter whether the stacks are used by multiple threads or a single thread. On a high level, it works by automatically detecting the use of a different stack and switching RCAP-stacks accordingly. In case a new stack is used, our instrumentation allocates a new RCAP-stack on demand.

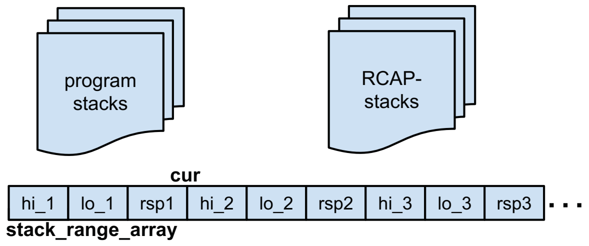

Figure 3: Multiple RCAP-stacks for multiple stacks

Figure 3 presents the overview of the scheme. For each stack used by the program, a RCAP-stack is maintained. The *stack_range_array* is a global data structure used to keep the range information for each stack. For the nth stack, its range information is kept in *hi_n* and *lo_n*, while the corresponding RCAP-stack pointer is *rsp_n*. Variable *cur* is a pointer to address metadata associated with the current stack and RCAP-stack.

The current stack range information is updated at instrumentation of calls, returns and RAstores, based on `%esp` value. However, if `%esp` is too far away from current stack range, a stack switch is identified. The *stack_range_array* is searched in order to look for the target stack information. If such a stack is not found, a new RCAP-stack is allocated on demand, and its metadata is also created in *stack_range_array*. After the right RCAP-stack is identified, pushes and checks of return addresses can be performed.

The threshold distance between `%esp` and current stack range needs to be set properly. It should be larger than maximum stack frame allocation size, yet small enough to distinguish with the accesses to another stack. We found the value of 256k typically works well, although a static analysis determining stack allocation sizes and other side information are useful for deciding a more accurate value.

Synchronization for global variable accesses may be needed but minimized. One limitation of our current approach is that it does not support stack reclaim and reuse. For example, if one stack is deallocated and the memory is reused for another execution context, it is not detected by simplying checking stack pointer change. However, this does not seem to happen often in practice.

4.3 RCAP-stack Protection

Since return capabilities are generated and consumed for control flow authentication, their integrity needs to be ensured. In other words, RCAP-stack which stores return capabilities should be protected. Otherwise, determined attackers could use vulnerabilities to corrupt both the program stack and RCAP-stack for control flow subversion.

Since our protection mechanisms are architecture-dependent, we will discuss them respectively.

4.3.1 Approach for x86-32

Our RCAP-stack protection for x86-32 is based on the segmentation feature on this architecture. Specifically, we set aside the top 96M of the 3GB user level address space for RCAP-stack use. Note that there could be multiple RCAP-stacks. Based on this arrangement, one additional segment is registered, which is supposed to be used by the program and excludes the safe region. Therefore, its base address is 0 and the limit is 0xb9ffffff. The original segment, which covers all user space, is used by our instrumentation instead. All our RCAP-stacks and their pointers are located in the protected memory.

Note that we chose the upper part of user space for safe region because we want to minimizes its compatibility impact for program code. A base address of 0 for the segment will make the program much less affected because pointers do not need to be adjusted. However, as program stack is usually mapped at a high address, special care needs to be taken if it overlaps our safe region. In this case, the loader is modified so that it relocates the stack at program start up time, before it sets up the additional segment for program use.

At program start up time, ds, es and ss are all set to use this segment. And any load to the segment registers are overridden. Only instrumentation code can use the original segment which includes safe region. To ensure the segments are not further modified, a system call policy is enforced.

4.3.2 Approach for x86-64

For architectures such as x86-64 which do not support segmentation-based protection, Software Fault Isolation (SFI) could be used to sandbox memory accesses. However, SFI can incur high overhead. In addition, the complexity of the instruction set makes identifying and sandboxing all memory accesses challenging. Therefore, we opt for an alternative route.

Our approach is based on RCAP-stack location randomization together with limited x86-64 segmentation functionality. Note that although in x86-64 cs, ss, ds, and es are all forced to 0 and the limit to 2^{64}, and memory accesses cannot be constrained, fs and gs can still have a nonzero base address. Therefore, we set up a segment for RCAP-stack with randomized location, and only reference it using fs in instrumentation code.

In general, two concerns could be associated with randomization based approach: brute force and information leak, which we discuss respectively.

In x86-64, addresses are 48 bits. Since the most significant bit for user space address is always 0, 47 bits are unknown to attackers. We assume the total size of all RCAP-stack is $128M = 2^{27}$ bytes, then the chance for a successful guess is 1 out of $2^{47-27} = 2^{20}$. In other words, there are 20 bits of entropy. Although this may be affected by program memory layout or other factors, and the available entropy could be smaller, the close-to 20 bit entropy would make brute force attacks hardly feasible, as false guesses would usually crash the program and alarms are raised for too many crashes.

The other issue is the possibility of leaking pointers to RCAP-stack. This is prevented because the pointer to RCAP-stack is never stored in unprotected memory. The only access is through fs register in instrumentation code.

5. Implementation

5.1 Static Analysis

The first step of static analysis is to extract candidate snippets. We utilized P$\mathbb{S}$I [58] for this purpose. Specifically, P$\mathbb{S}$I has a disassembly engine that is based on objdump and provides extra error recovery mechanism. It also builds a CFG for the code disassembled. We traversed the CFG backwards from each return instruction to collect code snippets up to N instructions, and we set N to 30 in our experiments.

For semantic analysis of candidate snippets, the semantics of each involved instruction is needed. We therefore lift up the assembly code into an intermediate representation (IR) so that the semantics are explicit. This is achieved by first translating assembly code into GCC's Register Transfer Language (RTL) IR, using a tool that reverses the GCC compilation process [25, 26, 23], and then flattening the RTL IR (tree representation) into a linear IR to simplify analysis. To compute the semantic equations, the equation for each instruction is first generated, and then the variables on the right hand side are repeatedly substituted (bound) with variables defined earlier, until only initial symbols present. Normalizations such as variable ordering and constant folding are performed during this procedure.

5.2 Binary Rewriting based Enforcement

Our shadow stack instrumentation is based on P$\mathbb{S}$I [58] and was implemented as its plugin. We chose P$\mathbb{S}$I primarily for two reasons. First, shadow stack needs to be built on top of CFI to be effective against ROP attacks, and P$\mathbb{S}$I offers CFI as a primitive. Second, P$\mathbb{S}$I is a platform for COTS binary instrumentations and works on both executables and shared libraries, and therefore aligns with our goal of instrumentation completeness.

Protecting the Dynamic Loader Since the dynamic loader ld.so is an implicit dependency for all dynamically linked executable, and is mapped to program address space to take charge of loading all other modules, it is also instrumented to prevent returns being misused. We made sure protected memory is set up before it is used by instrumentation.

Signal Handling The static analysis discussed in Section 3.3 is able to identify non-standard returns that consume return addresses stored by program code. However, return addresses can sometimes originate from the operating system. This is the case for UNIX signals. Once the OS delivers a signal to a process, it invokes the registered signal handler by switching context so that the user space execution starts at the first instruction of the signal handler. Prior to that, the OS puts the address of the sigreturn trampoline on the stack, which is to be used as the return address for the signal handler. Therefore, signal handler will "return" to the sigreturn trampoline, whose purpose is to trap back to the kernel. The kernel can proceed and revert user program execution with saved context. Since the returns for signal handlers (which are just normal functions) are also instrumented, if the corresponding return capabilities are not pushed onto RCAP-stack, signal delivery would cause false positives.

Fortunately, P$\mathbb{S}$I [58] already has a mechanism for signal handler mediations. The platform intercepts all signal register system calls such as signal(2) or sigaction(2) and registers wrappers as signal handlers. Once a wrapper function is invoked in case of a signal, it transfers control to the real signal handler after resolving its address. We use an updated version of wrapper code so that it pushes corresponding return capabilities to RCAP-stack.

6. Evaluation

We have extensively evaluated the key aspects of our system using different software on Linux and FreeBSD operating systems. We present our results and findings next.

6.1 Compatibility

Directory	Linux NSR #	Linux NSR module #	FreeBSD NSR #	FreeBSD NSR module #
/lib	9	4	7	2
/usr/lib	41	23	0	0
/bin	6	1	7	2
/sbin	6	1	4	1
/usr/bin	26	7	0	0
/rescue	N/A	N/A	182	91
/opt	28	7	N/A	N/A
total	116	42	213	98

Figure 4: Non-standard return (NSR) statistics

Module	OS	NSR Count
/lib/ld-2.15.so	Linux	2
/lib/i386-linux-gnu/libc.so.6	Linux	2
/lib/i386-linux-gnu/libgcc_s.so.1	Linux	4
/usr/bin/cpp-4.8	Linux	4
/usr/bin/g++-4.8	Linux	4
/usr/bin/gcc-4.8	Linux	4
/lib/libc.so.7	FreeBSD	3
/lib/libgcc_s.so.1	FreeBSD	4
/usr/bin/clang	FreeBSD	5

Figure 5: Non-standard return (NSR) in widely used modules

In this section, we evaluate the compatibility improvement of our approach. Since our static analysis pin-points non-standard returns, which are essentially causes of compatibility issues, we first present the static analysis results on statistics of identified non-standard returns. Then we provide a summary on our understanding of why these non-standard returns exist, and how they are used. Finally, we demonstrate the improved compatibility by testing our instrumentation on low-level and real world software.

6.1.1 Non-standard Return Statistics

We run our static analysis tool on executables and shared libraries from a Ubuntu 12.04 32 bit Linux desktop distribution, and a FreeBSD 10.1 32 bit desktop distribution. We have identified hundreds of non-standard return instances from different modules. Figure 4 shows the number of non-standard return instances and the modules containing them for different directories of Linux and FreeBSD.

To better understand the impact of non-standard returns to shadow stack compatibility, we need to further zoom in and see if they exist in widely used binary modules. Figure 5 shows the non-standard return instance number in some of the widely used modules.

6.1.2 Non-standard Return Summary

As discussed, return instructions are not necessarily used for subroutine exits, and they could be used in "novel" ways for different purposes. In this section, we summarize some of the most important reasons why non-standard returns are introduced, based on the found non-standard returns by our static analysis.

1. Programming language design and implementation

   In addition to subroutine abstraction, return instructions can also be used to implement other control flow abstractions such as coroutines or light-weight threads. Under these situations, they are used to transfer control between contexts, and therefore do not match calls.

2. Operating system design and implementation

   Operating systems also provide programmers various abstractions to ease their job. These abstractions may use return to implement control flow behavior across OS boundary. UNIX signals, as discussed, are probably the most prominent example in this category.

3. Software engineering practice

   In the engineering of some software constructs, programmers tend to make "clever" uses of assembly instructions. This also happens to return instructions. The dynamic function dispatch is a representative example of this.

Type	Software	Description
Low-level	libtask	Run a tcp proxy that uses user level threads API provided by libtask
Low-level	libunwind	Run a test program unwinding its own stack based on libunwind API
Real world	vim	Edit text file, search, replace, save
Real world	gedit	Edit text file, search, replace, save
Real world	mplayer	Play 10 mp3s
Real world	latex	Compile 10 tex files to dvi
Real world	python	Run pystone 1.1 benchmark
Real world	scp	Copy 10 files to server
Real world	evince	View 10 pdf files
Real world	wireshark	Capture ethernet packets for 10 min

Figure 6: Low-level and real world software testing

6.1.3 Testing Low-level and Real World Software

In order to further evaluate the compatibility of our approach, we tested with some low-level libraries and real world software. For each binary module of these software, we first run our static analysis to identify non-standard returns and RAstore instructions. The results are then feeded into the our instrumentation to generate hardened binaries. The instrumented software is finally executed for testing. For some of the programs that are not fully supported by PSI platform, we developed a Pintool for instrumentation, following the same approach.

Note that this evaluation is focused on identifying additional non-standard returns that are not pin-pointed by our static analysis. Therefore, the instrumentation is configured so that each identified control flow violation prints an alarm message about the violating instruction and the disallowed target. This helps us identify unknown non-standard returns should they execute.

Figure 6 shows the low-level and real world software we have tested, and how we tested them. No alarms are triggered with these tests, indicating our approach is compatible with them.

6.2 Protection

6.2.1 Security Analysis

Our system instruments all software modules including executables, shared libraries, and dynamic loader. Moreover, it protects all backward edges including both standard and non-standard returns. As compared to a static CFG, the intended control flow is enforced. Therefore, the completeness and precision of our system ensures the protection for all ROP attacks.

More advanced ROP attacks can leverage gadgets allowed by CFI to form payload. However, no matter how they are constructed, a single use of ROP gadget will be detected. Note that although JOP and COP gadgets are also used in advanced ROP attacks, the vast majority of them still rely on ROP gadgets [21, 16, 6, 22], and therefore can be defeated by our system.

Stack Pivoting. In ROP attacks, controlling the stack is of premiere importance. This is because, 1) fake return addresses need to be prepared on stack so that control flow can be repeated directed at attacker's will, and 2) the stack supplies data for gadgets.

Attackers basically have two choices to control the stack. The first is to *corrupt* the stack, usually through stack buffer overflow. The second is to *pivot* the stack, i.e., hijacking the stack pointer to point to attacker controlled data. Among these two, stack pivoting is more versatile because vulnerabilities other than buffer overflow could be used. It is also more convenient because there is no restrictions on location of corrupted memory.

Our system readily defeats ROP based on both stack corruption and stack pivoting. As the effectiveness for stack corruption is clear, we focus on the latter. Specifically, in a single RCAP-stack scheme, stack pivoting based ROP is blocked because the required return capabilities do not present on RCAP-stack. When multiple RCAP-stack are used, although stack pivoting could cause new RCAP-stack creation, security remains as no capabilities are yet issued.

Note that RCAP-stack protection is critical for defeating stack pivoting. This is because in addition to stack pivoting, the attacker could also craft and pivot a RCAP-stack by corrupting the RCAP-stack pointer. While previous solutions may be vulnerable to such attacks [17, 13], our system is resistant because RCAP-stack pointer resides in protectd memory as well.

TOCTOU Threats. For standard returns, our instrumentation pushes return capability onto RCAP-stack at the time of a call, i.e., the instant that return capability is issued. This is different from schemes that push return capability at function prologue [8, 13], which is some instructions later than the call. Unlike the latter, our system does not suffer from Time-Of-Check-to-Time-Of-Use (TOCTOU) attacks.

However, we note that our instrumentation does have a delay to store return capability in the case of a non-standard return: i.e., it happens at RAstore instruction, rather than return address generation instruction. This is due to limited data flow tracking of our analysis, and is not an issue when annotation is possible.

Storing return capability at a later time may give some window for attackers, because they can modify the generated return capability before its store on both stacks. However, attackers' ability for utilizing non-standard returns are greatly limited because of the following two reasons. First, CFI is still enforced as our base policy. Even if return capabilities for non-standard returns can be altered by attackers, it has to satisfy CFI at least, and therefore the forged capability can only grant transfer to instructions after calls. Second, as shown in Section 6.1.1, there are limited number of non-standard returns. Repeatedly corrupting return capabilities before store, effectively chaining such limited gadgets and bypassing CFI would be very difficult.

6.2.2 *Effectiveness for ROP Defense*

Although our approach by principle should defeat all ROP attacks, we evaluated its effectiveness with two real world ROP cases. Our first test is ROPEME [31], which exploits a buffer overflow vulnerability in a test program. The attack is two-staged. In the first stage the attack uses limited gadgets in the non-randomized executable code to leak out the base address of libc, so that gadgets in libc are available, and then dynamically generates a payload for chaining libc gadgets. In the second stage, stack is pivoted to this payload, and control is transferred to libc gadgets. Our defense blocked the attack at the first stage, because a backward edge control flow violation was identified when the vulnerable function returned with an overwritten return address.

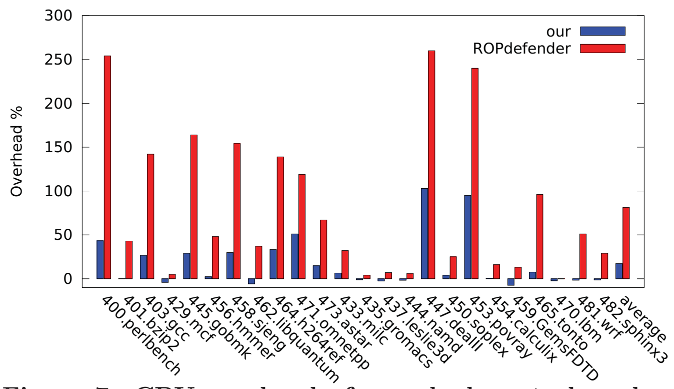

Figure 7: CPU overhead of our shadow stack and ROPdefender on SPEC 2006

Our second test is to protect a vulnerable Linux hex editor: HT Editor 2.0.20. A specially crafted long input could overflow the stack and lead to ROP attack [1]. Similarly, we detected the very first control flow violation and successfully defeated ROP attack.

6.3 Performance Overhead

We have measured the CPU overhead of our instrumentation on SPEC 2006 benchmark. We tested on a x86-32 Linux machine because it is the only environment currently supported by PSI [58]. For all the benchmarked programs, we transformed all involved executables and shared libraries. The results are presented in Figure 7, together with those from ROPdefender [17] as a comparison.

From Figure 7, we can see that the performance overhead of our system is about 17% on average, which is less than one fourth of that of ROPdefender, about 81%.

Similar to ROPdefender, a recent shadow stack implementation, Lockdown [41] is also based on dynamic binary instrumentation. It offers forward-edge control flow protection and has significantly improved performance. Over the common 22 SPEC 2006 programs evaluated, our system has an average 12% overhead, while Lockdown has about 24%.

7. Related Work

Bounds-Checking. The most comprehensive defense for memory corruption attacks are based on bounds-checking [29, 53, 54, 3, 34]. Unfortunately, these techniques introduce considerable overheads, while also raising significant compatibility issues. LBC [24] achieves lower overheads while greatly improving compatibility by trading off the ability to detect non-contiguous buffer overflows. Code pointer integrity (CPI) [30] significantly reduces overheads by selectively protecting only those pointers whose corruption can lead to control-flow hijacks.

Control-Flow Integrity. Control-Flow Integrity (CFI) [2] is a security policy that ensures program control flow always follow a pre-computed control flow graph(CFG). CFI can be effective against control flow hijacking attacks, under which control flow often deviate from the CFG. Therefore, the precision of the computed CFG decides the strength of the enforced policy.

The precision of CFG is dependent on the static analysis that computes it. For compiler based approaches, information such as types are available for computing a more precise CFG, therefore the enforced CFI is more fine-grained [35, 36, 50]. If CFI is applied for binaries, on the other hand, it is more coarse-grained because of the lack of higher level information [59, 57]. However, recent work has shown that advanced code reuse attacks can bypass not only coarse-grained CFI schemes, but also fine-grained ones [21, 16, 5, 10, 19].

To enforce an enhanced policy, CFI can be combined with techniques such as randomization [33] or cryptography [32]. Its strictness can also be improved using input [37, 14] or context [51] information. As a comparison, we focus on inferring intended control flow precisely using a static analysis.

ROP Defense. There are roughly two lines of work for ROP defense. The first concerns with enforcing control flow restrictions to detect ROP. This is different from CFI because the defense focuses on control flow properties of ROP rather than the integrity of each indirect control flow transfer. For example, kBouncer [40] and ROPecker [7] both use the Last Branch Record (LBR) feature of modern CPUs to detect the control flow chaining of short sequences of code. However, recent research [22, 6] has demonstrated that the techniques they use can be bypassed.

The second line of work aims to disrupt the availability of gadgets. GFree [38] is a compiler based technique that removes all unintended gadgets by replacing certain bytes from code with semantically equivalent instructions. It further restricts the use of intended code as gadgets by enforcing that returns must first experience function entries. As a comparison, our technique can be applied to all modules including third party libraries. The other choice that limits gadget availability is to use fine-grained randomization to break attacker's assumption about code layout. These randomizations can be applied at basic block level [52] or instruction level [39, 27, 18].

Approaches based on fine-grained randomization are vulnerable to JIT-ROP attacks [48] which leak program code and assemble ROP at runtime. Oxymoron [4] enables code sharing for randomized code, and provides a level of protection for JIT-ROP. However, it is shown to be bypassed [15]. Isomeron [15] integrates execution path randomization into fine-grained code randomization and is able to defend against JIT-ROP attack. However, the safe storage of selected call paths uses a similar structure to shadow stack, therefore sharing the same compatibility issues when calls and returns don't match. Readactor [12] is another effort that resists both direct and indirect memory disclosure attacks.

Shadow Stack. Shadow stack is the technique of maintaining and checking extra copies of return addresses on a separate stack. It is first used as a defense for stack smashing attacks [8, 9], and later being refined for ROP attacks [2, 17, 47, 42, 41, 58, 13], due to its benefits of precise control flow enforcement for backward edges. Shadow stack based ROP defenses can be roughly classified as dynamic instrumentation based approaches [17, 47, 42, 41], and static instrumentation based approaches [2, 58, 13]. While static shadow stack instrumentation needs CFI as a primitive to avoid being bypassed with a control flow transfer to unaligned code, dynamic approaches can rely on its property that each executed instruction is first instrumented. No matter which approach is taken, none of previous works have systematically resolved the compatibility problems that calls and returns do not necessarily match.

8. Conclusions

In this paper, we presented a principled approach for ROP defense. Our approach accurately infers and enforces program intended control flow, which breaks one mandatory requirement for ROP: repeatedly subverting control flows. To that end, we developed static analysis techniques and utilized static binary instrumentation for enforcement. Experimental evaluations have shown that our approach provides precise control flow guarantees, yet efficient and compatible with real world applications.

9. References

[1] HT editor 2.0.20 - buffer overflow (ROP PoC). https://www.exploit-db.com/exploits/22683/.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Computer and Communications Security*, 2005.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[4] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[5] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[6] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[7] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Annual Network and Distributed System Security Symposium*, 2014.

[8] T.-c. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, 2001.

[9] T.-c. Chiueh and M. Prasad. A binary rewriting defense against stack based overflow attacks. In *USENIX Annual Technical Conference*, 2003.

[10] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Computer and Communications Security*, 2015.

[11] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. *SIGARCH Comput. Archit. News*, 2005.

[12] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Security and Privacy*, 2015.

[13] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security*, 2015.

[14] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *Annual Design Automation Conference*, 2015.

[15] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Annual Network and Distributed System Security Symposium*, 2015.

[16] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of

coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[18] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[19] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Computer and Communications Security*, 2015.

[20] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference*, 2014.

[21] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming control-flow integrity. In *IEEE Security and Privacy*, 2014.

[22] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[23] N. Hasabnis. *Automatic Synthesis of Instruction Set Semantics and its Applications.* PhD thesis, Stony Brook University, 2015.

[24] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *ACM CGO*, 2012.

[25] N. Hasabnis and R. Sekar. Automatic generation of assembly to IR translators using compilers. In *Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2015.

[26] N. Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. Technical report, Secure Systems Lab, Stony Brook University, 2015.

[27] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. ILR: Where'd my gadgets go? In *IEEE Security and Privacy*, 2012.

[28] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Annual Network and Distributed System Security Symposium*, 2014.

[29] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, 1997.

[30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[31] longld. Payload already inside: Data reuse for rop exploits. https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf.

[32] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. CCFI: Cryptographically enforced control flow integrity. In *ACM Computer and Communications Security*, 2015.

[33] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Annual Network and Distributed System Security Symposium*, 2015.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *PLDI*, 2009.

[35] B. Niu and G. Tan. Modular control-flow integrity. In *ACM Programming Language Design and Implementation*, 2014.

[36] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Computer and Communications Security*, 2014.

[37] B. Niu and G. Tan. Per-input control-flow integrity. In *ACM Computer and Communications Security*, 2015.

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

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Security and Privacy*, 2012.

[40] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[41] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2015.

[42] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE Security and Privacy*, 2012.

[43] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Annual Network and Distributed System Security Symposium*, 2015.

[44] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *IEEE/ACM Code Generation and Optimization*, 2008.

[45] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Security and Privacy*, 2015.

[46] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Computer and Communications Security*, 2007.

[47] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[48] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Security and Privacy*, 2013.

[49] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Security and Privacy*, 2013.

[50] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[51] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *ACM Computer and Communications Security*, 2015.

[52] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Computer and Communications Security*, 2012.

[53] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. 2004.

[54] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *ACM ASIACCS*, 2010.

[55] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Computer and communications security*, 2011.

[56] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Annual Network and Distributed System Security Symposium*, 2015.

[57] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[58] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *ACM Virtual Execution Environments*, 2014.

[59] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.