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 on COTS binaries. Our experimental evaluation shows that our approach enforces a stronger policy, while offering better compatibility and performance as compared to previous research. Our prototype is compatible with many real-world and low-level programs. On SPEC 2006 benchmark program, it adds just 4% overhead above the base overhead of 13% imposed by PSI, the platform used in our implementation.

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.

*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

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 [17, 14] — 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 [14].

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 [23, 43, 37, 48], and some of them are already being deployed in production compilers such as gcc and llvm [43]. This factor can greatly reduce the threat of code reuse attacks based on repeated subversion of forward edges, e.g., call-oriented programming [6, 39]. 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, 12]. 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 returns (See Section 2.1) 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 deter practical deployment of shadow stack approaches.

- *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

Our approach is based on the following simple policy:

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

With a static interpretation of "intention", 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 do 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).

1.2 Contributions

We make the following contributions in this paper:

- *Static analysis to handle non-standard returns:* While the intended returns of call instructions are obvious, nontrivial applications include many non-standard returns that don't match any calls. Unlike previous approaches that relied on manual annotations to handle them, we present an automated static analysis technique that identifies (a) non-standard returns, and (b) the intended targets of these returns. Our analysis has been sufficient to handle SPEC 2006 benchmarks as well as several complex applications studied in our evaluation.
- *Support for diverse threading mechanisms:* In addition to non-standard returns, multi-threaded programs pose a challenge for shadow stack mechanisms. This is because shadow stacks have to be created and switched in sync with thread creation and switching. This challenge is exacerbated in complex applications because they may use a variety of threading mechanisms. We present a unified, threading-mechanism-independent approach for maintaining per-thread RCAP-stack, and transparently switching to the right RCAP-stack at runtime, based on the value of stack pointer at the point of return.
- *Strict enforcement:* By discovering and always pushing the intended target on RCAP-stack, we avoid the need for "whitelisting" return instructions. By subjecting all returns to a strict policy, we take away an attacker's capability to abuse even a single return instruction in the protected application.
- *In-depth evaluation:* Our evaluation demonstrates excellent compatibility as well as protection against attacks. The overhead of our solution is significantly lower than previous techniques: just 3% over the base 14% overhead posed by our platform P𝕊I on SPEC 2006 benchmarks.

2. Background and Threat Model

2.1 ROP Attacks

Return-Oriented Programming (ROP) [40] is the most prevalent form of code-reuse attack. It makes use of "gadgets," i.e., existing code snippets ending with return instructions. A gadget used in a 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 on variable-length instruction-set architectures such as the x86.

To carry out a ROP-attack, it is necessary to subvert a program's intended control flow. We develop an effective policy to break this step and hence prevent ROP attacks.

2.2 Control-Flow Integrity (CFI)

Control-flow integrity is an important primitive for *secure* static instrumentation [50, 47], 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 within (or immediately following) an inserted instrumentation snippet. For this reason, we build our defense, which is based on static binary instrumentation, on a platform that already implements CFI, specifically, the P𝕊I platform [50].

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 assume DEP is enabled on the victim system and therefore ROP is a necessity for payload construction. We also assume that ASLR is 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

As discussed earlier, we focus exclusively on return instructions. We do not attempt to further improve the (coarse-grained) BinCFI policy [51] enforced on the remaining branch types by our implementation platform, namely, P𝕊I [50].

The first task in enforcing a stronger policy on returns is to precisely infer program-intended control flow for each of them. We develop a static analysis for this purpose. Specifically, our analysis identifies instructions that push addresses that may later be used as the target for a return instruction. As a fallback option, static analysis may be augmented with manual annotations, but we have not had to do this so far.

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

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

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.

3.1 Calls

Most call instructions are used for function invocations and therefore express an intent 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 is computed.

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

Returning to a code location is permitted only if the program possesses the capability to do so. We check RCAP-stack for this capability. Typically, this capability originates at the most recent call instruction, but there are instances where the return address is pushed by other means. We call such returns as *non-standard returns*.

One example of a 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[1], as shown in Figure 1.

Note that while this non-standard return achieves the effect of an indirect jump, it does so without using any register (other than the stack pointer), and moreover, deallocates the memory location used to store the target address.

As discussed in the next section, there are a number of such non-standard returns, scattered in different modules. Moreover, unlike a call, whose intended return address is its successor, the intended target of non-standard return is not immediately obvious. These factors motivate the static analysis described below.

3.3 Static Analysis of Non-standard Returns

The distinction between a standard and non-standard return is the return address being used. The return address used by a *standard return* is pushed by the call instruction in its caller, and not modified in the callee. In contrast, return address used by a *non-standard return* is written to the return address stack slot by a non-call instruction. Based on this observation, we develop a static analysis that consists of four main steps as discussed below.

Candidate snippet extraction. After a binary module is disassembled, we build its CFG. We then perform a backward scan on the CFG starting from each return instruction, and going back by n instructions, with $n = 30$ in our implementation. These snippets are our candidates for analysis.

Each such snippet may contain multiple execution paths to the return instruction. We analyze each path separately, as this enables more accurate analyis. In particular, this approach avoids approximations that result from least upper bound operations needed to handle path merges. However, this approach introduces two problems. First, loops can lead to an unbounded number of paths. We only consider paths corresponding to zero and one iteration of such loops. As a result, we may fail to discover some instances where an instruction inside a loop pushes a return address on the stack. In theory, this could lead to a compatibility problem, but in practice, it is very unlikely that such instructions occur within a loop body. The second difficulty is that it is theoretically possible for a single instruction I to participate in two distinct paths such that in the first path, I pushes a value on the stack that would be used by the return instruction at the end of the snippet, while it does not do so in the second path. Note that this (unlikely) scenario does not lead to an incompatibility: if the second execution path were to be taken at runtime, the return capability pushed by I would simply not be used.

Semantic analysis. The second step is to analyze the semantics of each snippet by performing an abstract interpretation using an abstract domain similar to that used in Reference [38]. 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, the abstract value of each register (or memory location) will consist of simple expressions consisting of constants and initial register values. Since we are analyzing each execution path separately, these expressions rarely involve approximations. Our analysis includes a simple procedure for maintaining these expressions in a canonical form, thereby enabling equivalent expressions to be recognized in most instances.

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

[1]The argument `0xc` to the `ret` instruction specifies the number of additional bytes that should be popped off the stack.

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 values used by the return. We call such an instruction as an *RAstore*. Such an instruction can be identified from the contents of memory and registers computed by our static analysis after each instruction in the snippet.

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 [15, 50, 12]. However, manual approaches are not scalable, and/or can lead to false positives on large and complex software. Figure 2 illustrates some of the more prominent real-world non-standard returns identified by our static analysis. In this figure, upper case register names (e.g., EAX) denote initial symbolic values, while lower case ones (e.g., eax) denote the current contents of registers or memory. For easier illustration, each code snippet is simplified to only include the last basic block. We omit the effects on 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. Our analysis indicates that the 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.

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 [15, 12], only the first and third of these four examples have

Code Snippet	Semantics Equations
;; #1 /lib/ld-2.15.so	eax = *(ESP+8)
0x146b0 popl %edx	edx = *ESP
0x146b1 movl (%esp),%ecx	ecx = *(ESP+4)
0x146b4 movl %eax,(%esp)	esp = ESP + 4
0x146b7 movl 0x4(%esp),%eax	*(ESP+4) = EAX
0x146bb ret $0xc	ra = *esp
	= *(ESP+4) = EAX
;; #2 /lib/i386-linux-gnu/libc.so.6	
0x3fa73 movl 0x4(%esp),%eax	eax = *(*(ESP+4)+64)
0x3fa77 movl 0x60(%eax),%ecx	edx = *(*(ESP+4)+56)
0x3fa7a fldenvl (%ecx)	ecx = *(*(ESP+4)+60)
0x3fa7c movl 0x18(%eax),%ecx	ebx = *(*(ESP+4)+52)
0x3fa7f movl %ecx,%fs	esi = *(*(ESP+4)+40)
0x3fa81 movl 0x4c(%eax),%ecx	edi = *(*(ESP+4)+36)
0x3fa84 movl 0x30(%eax),%esp	ebp = *(*(ESP+4)+44)
0x3fa87 pushl %ecx	esp = *(*(ESP+4)+48)-4
0x3fa88 movl 0x24(%eax),%edi	*(*(*(ESP+4)+48)-4)
0x3fa8b movl 0x28(%eax),%esi	= *(*(ESP+4)+76)
0x3fa8e movl 0x2c(%eax),%ebp	ra = *esp
0x3fa91 movl 0x34(%eax),%ebx	= *(*(*(ESP+4)+48)-4)
0x3fa94 movl 0x38(%eax),%edx	= *(*(ESP+4)+76)
0x3fa97 movl 0x3c(%eax),%ecx	
0x3fa9a movl 0x40(%eax),%eax	
0x3fa9d ret	
;; #3 /lib/i386-linux-gnu/libgcc_s.so.1	
0x154cb movl %esi,%ecx	eax = *(EBP-20)
0x154cd movl %edi,	
0x4(%ebp,%esi,1)	edx = *(EBP-16)
0x154d1 addl $0x10,%esp	ecx = ESI+EBP+4
0x154d4 leal 0x4(%ebp,%ecx,1),%ecx	edx = *(EBP-12)
0x154d8 movl -0x14(%ebp),%eax	esi = *(EBP-8)
0x154db movl -0x10(%ebp),%edx	edi = *(EBP-4)
0x154de movl -0xc(%ebp),%ebx	ebp = *EBP
0x154e1 movl -0x8(%ebp),%esi	esp = ESI+EBP+4
0x154e4 movl -0x4(%ebp),%edi	*(ESI+EBP+4) = EDI
0x154e7 movl 0x0(%ebp),%ebp	ra = *esp
0x154ea movl %ecx,%esp	= *(ESI+EBP+4)
0x154ec ret	= EDI
;; #4 /usr/lib/libunwind-setjmp.so	eax = EDX
0x674 pushl %eax	esp = ESP-4
0x675 movl %edx,%eax	*(ESP-4) = EAX
0x677 ret	ra = *esp
	= *(ESP-4) = EAX

Figure 2: Code snippets and their analysis results

seen manual handling [15]. 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. If this assumption were to be violated, we can address it by strengthening the analysis, or using manual annotations. As mentioned before, we have not had to do this so far in our implementation.

4. Enforcing Intended Control Flow

In this section, we describe our approach for enforcing intended control-flow using static binary instrumentation. A key component of our design is its support for standard as well as "non-standard" threading mechanisms in order to provide better compatibility with a wide range of software. Finally, we describe the protection of the RCAP-stack to ensure that the same mechanisms used to corrupt the main stack cannot corrupt the RCAP-stack.

4.1 Instrumentation-based Enforcement

Intended control flow enforcement is realized by instrumenting calls, RAstores and returns. Both calls[2] and RA-

[2]As discussed earlier, we avoid instrumenting calls that are determined never to return.

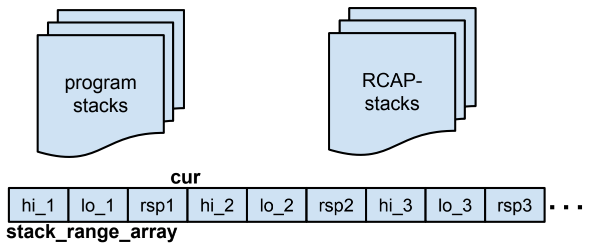

Figure 3: Multiple RCAP-stacks for multiple stacks

stores are instrumented in the same manner: a copy of the address being stored on the main stack is also pushed on RCAP-stack.

Return instructions are instrumented to check the RCAP-stack for the corresponding capability. Note that due to normal program behaviors such as stack unwinding, the required return capability may not always be located at the top of RCAP-stack. Similar to previous shadow stack proposals, our design also pops non-matching capabilities from the top of RCAP-stack until a capability that matches the target location of the return is encountered. If such a capability is never found, then a policy violation is reported and program execution aborted.

4.2 Multi-threading Support

The operating system allocates a separate stack for each OS-visible thread. In addition, some applications may maintain multiple stacks at the user-level. This may be done to implement user-level threads (e.g., using `setcontext/getcontext` family of system calls) , or for purposes such as signal handling (e.g., using `sigaltstack`). Regardless of the mechanism used, it is necessary to maintain a unique RCAP-stack for each stack used by the application.

If the only threading mechanism in use consisted of OS-visible threads, then there is a simple solution for maintaining one RCAP-stack per main stack: the typical solution used in previous shadow scheme techniques (e.g., [50]) is to store the shadow stack pointer in thread-local storage (TLS). However, this technique won't work correctly in the presence of user-level threading mechanisms. We therefore develop a simple but uniform technique for supporting various context-switching mechanisms transparently. This technique operates by recognizing changes to the stack pointer value, and switching to the corresponding RCAP-stack.

Figure 3 illustrates the design. 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. A threshold value needs to be defined for this scheme — we found that 256K worked well in experiments. 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.

Maintaining and updating global data structures can be expensive because of the need to lock them before access. To reduce the need for locking, we cache the range of the current stack in TLS. A pointer to the current RCAP-stack is also stored in TLS. Since most calls, returns and RAstores don't involve a switching of stacks, this design allows them to be handled without accessing global storage, and hence without the need for locking. When a switch is recognized, the cached information is propagated from TLS to global data structures, and the switching performed as before.

Although our approach is simple and can generally support diverse context-switching mechanisms, it has some limitations: one issue is that although we can detect the use of a new stack and associate a newly allocated RCAP-stack with it, there is no way to determine if there are valid return addresses already on the stack. Our solution is to require programmer annotation for constructs that exhibit this behavior. In our experiments, this annotation has been needed for just one operation, namely, the call to `makecontext`. Second, if a stack is freed and reallocated, then we may end up using a stale RCAP-stack. This won't lead to false positives, but very slightly increases the attacker's abilities: attacks can now return to one of the stale return addresses on the RCAP-stack. Third limitation concerns JIT code: since our implementation is based on off-line static analysis, it does not currently support runtime-generated code.

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.

We used the same approach as described in CFCI [52], which has also been implemented on our platform PSI. In short, the protection mechanisms are architecture-dependent. For x86-32, we rely on segmentation for efficient protection, and for x86-64, a randomization based approach is used. The randomization approach ensures that the location of RCAP-stack cannot be leaked.

5. Implementation

5.1 Static Analysis

The first step of static analysis is to extract candidate snippets. We utilized PSI [50] for this purpose. Specifically, PSI has a disassembly engine that is based on objdump, and adds a layer of error detection and correction over it. It also builds a CFG for the code disassembled. We traversed the CFG backwards from each return instruction to collect code snippets that were 30 instructions long.

For our static analysis, we need to accurately model the semantics of each instruction. Specifically, we utilized a tool by Hasabnis et al [21, 19] that lifts assembly to GCC's intermediate language called RTL. This tool is driven by GCC's architectural specifications, and can hence support all x86 instructions, as well as other ISAs supported by GCC.

Our lifting enables the semantics of each instruction to be captured using a handful of RTL operations, e.g., arithmetic operations, memory dereferencing, and assignment. As a result, our static analysis can be implemented in an architecture-neutral fashion. Moreover, is can side-step the complexities posed by large instruction sets such as the x86.

RTL is a tree-structured language. To simplify analysis, we flatten RTL into a sequence of triples, each consisting of a destination and up to two source operands. Our static analysis is performed on these triples. Since we analyze single execution paths, the main step in the static analysis is to substitute each register or memory location by the expression representing its previously computed value. This expression is maintained in a canonical form by defining an ordering on variables, and by performing constant-folding and other arithmetic simplifications.

5.2 Binary Rewriting based Enforcement

Our shadow stack instrumentation is based on P𝕊I [50] and was implemented as a plugin. We chose P𝕊I primarily for two reasons. First, shadow stack needs to be built on top of CFI to be effective against ROP attacks, and P𝕊I offers CFI as a primitive. Second, P𝕊I 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 executables, it is also instrumented to prevent returns from being misused. We ensured that memory protection for RCAP-stack 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, P𝕊I [50] already has a mechanism for signal handler mediation. The platform intercepts all signal handler registrations (using `signal` and `sigaction` system calls) and registers wrappers for the signal handlers. Once a wrapper function is invoked by the OS, it transfers control to the real signal handler after resolving its address. We use an updated version of wrapper code so that it pushes the corresponding return capabilities to RCAP-stack. (The wrapper code is not instrumented, and the CFI policy configured to ensure that it cannot be invoked by the application.)

5.3 Optimizing returns

Our shadow stack is built on top of a binary instrumentation system that requires code pointer translation. In particular, code pointers point to original code section, while the instrumented code resides in a different section. As a result, code pointer values need to be translated to the corresponding code locations in the instrumented code. This step, called address translation, is a significant source of runtime overhead because it requires a hash table lookup. To improve the performance, we performed an optimization that has also been used in some previous research works [36]: push both the original address and translated address on the shadow stack for each call. At the time of return, we first compare the return address on the main stack with the original address on the shadow stack, and if they match, return to the translated address on the shadow stack.

For calls, the translated return address is simply the address of the instruction following the call instruction. However, RAstores push code pointers on the stack, so there is no way to avoid address translation for them. Rather than eagerly performing address translation at the RAstore, we simply push a null value as the translated address. At a return instruction, if the translated address has a null value, we perform address translation at that point.

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 returns in common modules

6. Evaluation

We evaluated the key aspects of our system using a wide range of software on Linux and FreeBSD operating systems. Below, we present our findings and results.

6.1 Compatibility

In this section, we evaluate the compatibility improvement offered by our approach. We first present statistics on the identification of non-standard returns, together with an explanation for their prevalence. We then demonstrate the improved compatibility by testing our instrumentation on low-level and real-world software.

6.1.1 Non-standard Return Statistics

We ran our static analysis tool on executables and shared libraries from an 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 prevalence of non-standard returns in some of the widely used modules.

6.1.2 Non-standard Return Summary

Type	Software	Size	Description
Low-level	libunwind	1.9	Run a test program unwinding its own stack based on libunwind API
Low-level	libtask	2.0	Run a tcp proxy that uses user level threads API provided by libtask
Real-world	scp	2.1	Copy 10 files to server
Real-world	python	6.7	Run pystone 1.1 benchmark
Real-world	latex	7.8	Compile 10 tex files to dvi
Real-world	vim	9.1	Edit text file, search, replace, save
Real-world	gedit	22	Edit text file, search, replace, save
Real-world	evince	26	View 10 pdf files
Real-world	mplayer	46	Play 10 mp3s
Real-world	wireshark	58	Capture packets for 10 min

Figure 6: Low-level and real-world software testing

In this section, we summarize some of the most common reasons for the prevalence of non-standard returns, based on an analysis of our static analysis results.

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. Optimization "tricks"

   In the engineering of some software constructs, programmers tend to make "clever" uses of assembly instructions. This also happens to return instructions.

6.1.3 Testing Low-level and Real-World Software

In order to further evaluate the compatibility of our approach, we tested it with some low-level libraries and real-world software. For each binary module tested, we first ran our static analysis to identify non-standard returns and RAstore instructions. The results are then fed into our instrumentation module to generate hardened binaries. The instrumented software is finally executed for testing. For multi-threaded programs used in this evaluation, we used Pin [28] for our testing.

Figure 6 shows the low-level and real-world software we have tested, and how we tested them. The "Size" column specifies the total mapped code size (in MB) of all modules of the program. No incompatibilities were found on any of these programs, demonstrating that our approach works well even on low-level software. The total size of all software tested in this evaluation is almost 200MB.

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. Return capabilities greatly restrict the scope of attacks possible. A coarse-grained CFI permits any return to target any of the instructions following a call in a program. In contrast, our approach limits return to one of the return addresses that are already on the RCAP-stack. Moreover, each time an attack makes use of a return address other than the top entry on RCAP-stack, the intervening entries are popped off, thus further reducing the choice of possible targets for the next return.

Note that although JOP and COP gadgets can be used in advanced code-reuse attacks, the vast majority of them still rely on ROP gadgets [17, 14, 6, 18], and therefore can be defeated by our system.

Stack Pivoting. In ROP attacks, controlling the stack is the most important goal of the attacker. This is because, (a) fake return addresses need to be prepared on stack so that control flow can be repeatedly redirected in the manner chosen by the attacker, and (b) the stack supplies the data used in ROP computation.

Attackers basically have two choices to control the stack. The first is to *corrupt* the stack, usually through a stack buffer overflow. The second is to *pivot* the stack, i.e., hijack 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 the entire stack could be controlled, without being limited by factors such as the location of the vulnerable buffer, or the maximum length of overflow.

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 won't be present on RCAP-stack. When multiple RCAP-stacks are used, although stack pivoting could cause new RCAP-stack creation, this does not compromise security as the new RCAP-stack starts out with zero return capabilities on it.

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 an RCAP-stack by corrupting the RCAP-stack pointer. While previous solutions may be vulnerable to such attacks [15, 12], our system is resistant because RCAP-stack pointer resides in protectd memory as well.

TOCTTOU 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, 12], and hence provide a (narrow) window for TOCTTOU 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, attacker capabilities for utilizing non-standard returns is 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

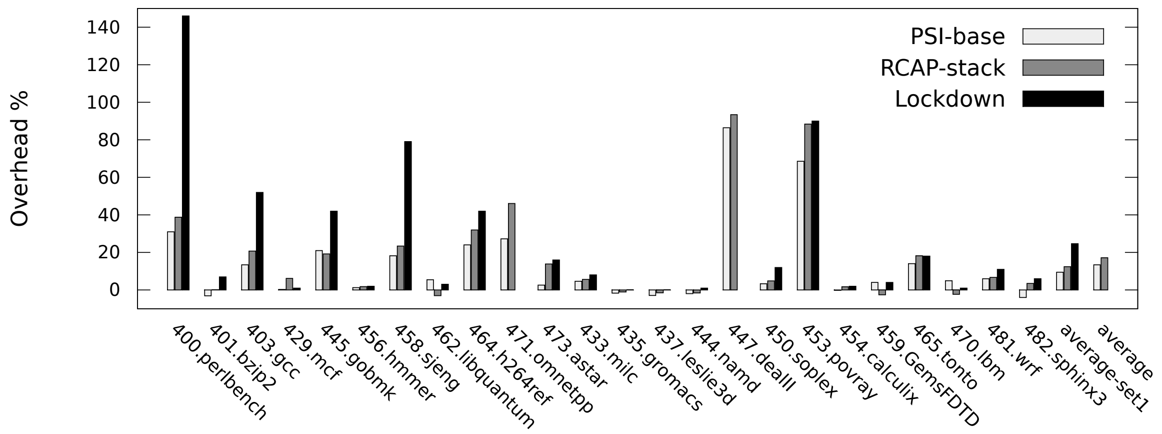

Figure 7: CPU overhead of shadow stack systems on SPEC 2006

non-standard returns. Repeatedly corrupting return capabilities before store, effectively chaining such limited gadgets and bypassing CFI would be very difficult.

6.2.2 *Experimental Evaluation of ROP Defense*

We evaluated the effectiveness of our approach using two real-world ROP attacks. Our first test was the ROPEME attack [27], which exploits a buffer overflow vulnerability in a test program. The attack is two-staged. In the first stage, the attack uses a limited set of gadgets in non-randomized executable code to leak out the base address of `libc`. This enables the attacker to bypass ASLR as it relates to targeting gadgets in `libc`. In the second stage of the attack, ROPEME uses a payload consisting of a chain of `libc` gadgets. The 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.

Our second test was 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]. As with the first attack, we detected the very first control flow violation and successfully defeated this ROP attack as well.

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 [50]. For all the benchmarked programs, we transformed all involved executables and shared libraries. The results are presented in Figure 7, where an empty bar in the histogram indicates an unavailable performance number.

We compare our performance with that of our base platform P𝕊I [50] and Lockdown [35], a recent dynamic instrumentation based shadow stack implementation. From Figure 7, we can see that the performance overhead of our system is about 17% on average. Our optimization (Section 5.3) accelerates several control-flow intensive benchmarks such as 429.mcf and 447.dealll by 9% and 403.gcc, 458.sjeng, 471.omnetpp, and 453.povray by 5%. For the common set of programs we had with Lockdown, our overhead is 13% while theirs is about 24%.

Parallel shadow stack [12] achieves lower overhead by employing a variety of optimizations. They report overheads in the range of 3.7% to 4.6%. Their approach does not operate on binaries, but instead, on the assembly code produced by a compiler. As a result, they avoid the overhead of address translation. In addition, they do not enforce CFI. Considering these are the two major source of overhead for the P𝕊I platform we used, our added overhead of 4% makes our performance comparable to theirs.

7. Related Work

Bounds-Checking. The most comprehensive defense for memory corruption attacks is based on bounds-checking [24, 45, 46, 3, 29]. Unfortunately, these techniques introduce considerable overheads, while also raising significant compatibility issues [42]. LBC [20] achieves lower overheads while greatly improving compatibility by trading off the ability to detect non-contiguous buffer overflows. Code pointer integrity [26] significantly reduces overheads by selectively protecting only those pointers whose corruption can lead to control-flow hijacks.

Control-Flow Integrity. Control-flow integrity can significantly limit an attacker's choices for diverting control flow [2, 51, 49]. The modest performance overheads of these techniques can be further significantly reduced using modern CPU features such as the Last Branch Record (LBR) [34, 7]. The downside, however, is a looser policy that can be more easily bypassed. Indeed, several researchers have shown that in general, coarse-grained CFI techniques canbe defeated [17, 14, 18, 6]. The biggest culprit behind this weakness is the policy applied for returns, which typically allows control to go back to any instruction that follows a call. Shadow stack schemes such as ours eliminate this weakness. The second major source of CFI weakness is the large number of valid targets for indirect calls, especially in C++ programs. Forward-edge CFI techniques [43] can significantly narrow down these targets using information such as function signatures.

G-Free [32] is a novel ROP defense technique that eliminates *unintended* gadgets. It requires programs to be recompiled, and introduces a rewriting phase on the compiler's assembly code output. This phase ensures that the return opcode will not occur in the middle of valid instructions. To guard against the use of gadgets ending with intended return instructions, G-Free "encrypts" return addresses by XOR-masking with a secret key, but this scheme is weak against adversaries that can leak return addresses on the stack. Shadow stack approaches avoid this weakness.

Code Randomization. These techniques may be aimed at randomizing the actual instruction content, or simply their locations. Instruction-space randomization (ISR) [25, 4] is a systematic technique for randomizing instruction content by encrypting them. In practice, it has proved difficult to develop encryption schemes that provide adequate performace for ISR, while resisting attacks to reverse-engineer the key. Consequently, recent efforts [33] have avoided encryption, instead replacing an instruction with a functionally equivalent one, or using techniques such as register reassignment.

Layout randomization, e.g., ASLR, can block ROP attacks by making gadget locations unpredictable. Since coarse-grained ASLR can often be defeated by leaking a single code or data address, Bhatkar et al [5] developed fine-grained techniques that randomly re-orders all data objects as well as functions. Note that purely compile-time permutations are ineffective, unless existing binary and patch distribution models are changed: otherwise, the attacker will have access to the same randomized binary as the victim. For this reason, Bhatkar et al [5] use compile-time code transformations to generate the information required for load-time randomization. More recent works such as binary stirring [44] and ILR [22] can operate without compiler help, and moreover, can reorder basic blocks rather than functions.

JIT-ROP attacks [41], which rely on memory disclosure vulnerabilities to scan code sections, can defeat fine-grained randomization by discovering gadgets at runtime. Several recent works [13, 11] have developed defenses to JIT-ROP.

In general, randomization techniques can effectively block unintended gadgets, but disrupting the use of gadgets beginning at legitimate indirect control-flow targets (i.e., targets permitted by CFI schemes) can be difficult. One reason is that programs store such targets in memory, thereby making them vulnerable to information disclosure. Indeed, most binary instrumentation techniques don't change the original values of code pointers, so information disclosure is not even necessary with these schemes. A second reason is that the targeted code has legitimate invokers that expect a certain semantics from it, including the contents of registers, caller-callee conventions, and so on. This significantly limits the scope of what is achievable using randomization without breaking legitimate functionality. In any case, it is safe to say that shadow stack schemes such as ours enforce as strong a policy on returns as any other randomization scheme, and moreover, provide deterministic rather than probabilistic protection.

Shadow Stack. Shadow stack schemes [16, 8] were first proposed as a defense for stack smashing attacks. However, only the legitimate returns were checked, so ROP attacks using unintended returns were possible. CFI enforcement, which prevents the use of unintended instructions, provides one way to block this attack avenue. A second approach, used in DBT-based techniques (e.g., ROPdefender [15]), is to instrument *all* returns before their execution.

The benefits of shadow stack for strengthening CFI [2], and to defeat ROP, have long been recognized. However, its practical deployment has been limited by the prevalence of non-standard returns that violate shadow stack checks. While RAD [8] addressed the cases of `longjmp` and signals, ROPdefender [15] identified two other non-standard uses: C++ exceptions and lazy-binding of calls to shared library functions. It handled them by manually identifying instructions that save a return address on the stack, and pushing a copy on the shadow stack. Thus, their runtime policy follows the return capability model used in our approach.

A drawback of ROPdefender was its significant runtime overhead. Zhang et al [50] discuss how dynamic binary instrumentation techniques, while displaying good performance on SPEC benchmarks, tend to perform poorly on large, real-world applications. Being based on static instrumentation, Zhang et al were able to achieve significantly better performance than ROPdefender.

Lockdown [35] is a recent effort combining shadow stack and CFI in dynamic instrumentation, while focusing on reducing runtime overhead. However, unlike our approach, they do not focus improving compatibility. Moreover, as disscussed earlier, on the common benchmarks, our overhead is about half of theirs.

Dang et. al surveyed existing shadow stack systems and designed a "parallel shadow stack" scheme [12] to eliminate the need for shadow stack pointer save and restore. They avoided register clobbers in their intrumentation, applied peephole optimizations, and achieved great performance. However, this comes with some trade-offs on security. In fact, StackDefiler [10] describes an attack that leaks shadow stack address. As a comparison, we maintain strong security by enforcing CFI, protecting RCAP-stack, and precisely managing return capabilities.

In summary, the primary contributions of this paper over previous works are (a) the development of a systematic approach for identifying and handling non-standard returns without the need for manual effort, (b) demonstrating that this approach can handle complex and low-level binaries, and (c) achieving low overheads.

Binary vs Source-code Based Defenses. Most of the techniques discussed above, including bounds-checking [24, 45, 46, 3, 29, 20, 26], fine-grained CFI [43, 30, 31], and many fine-grained randomization techniques [5, 11] and others [32], are based on compilers and operate only on source code. With such approaches, it is difficult to protect low-level code that uses inline assembly, as well as third-party code that only available in binary form. Unfortunately, security is usually dependent on the "weakest link," and even one unprotected module can render the defense bypassable. In contrast, binary-based defenses extends to all code, regardless of the programming language or the compiler.

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 principles, implementations, and applications. *ACM TISSEC*, 2009.

[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] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS*, 2003.

[5] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, 2005.

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

[8] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, 2001.

[9] T. Chiueh and M. Prasad. A binary rewriting defense against stack based overflows. In *USENIX ATC*, 2003.

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

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

[12] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and canaries. In *ASIACCS*, 2015.

[13] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[14] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[15] L. Davi, A. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[16] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security*, 2001.

[17] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[18] 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*, 2014.

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

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

[21] N. Hasabnis and R. Sekar. Automatic generation of assembly to IR translators using compilers. In *AMAS-BT*, 2015.

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

[23] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *NDSS*, 2014.

[24] 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.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, 2003.

[26] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[27] 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.

[28] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

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

[30] B. Niu and G. Tan. Modular control-flow integrity. In *PLDI*, 2014.

[31] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *CCS*, 2014.

[32] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *ACSAC*, 2010.

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

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

[35] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA*, 2015.

[36] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE S&P*, 2012.

[37] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *NDSS*, 2015.

[38] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

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

[40] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

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

[42] L. Szekeres, M. Payer, T. Wei, and R. Sekar. Eternal war in memory. *S&P Magazine*, 2014.

[43] 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*, 2014.

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

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

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

[47] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[48] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *NDSS*, 2015.

[49] 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 S&P*, 2013.

[50] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *VEE*, 2014.

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

[52] M. Zhang and R. Sekar. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*, 2015.