|galois|

GALOIS-17-01

Code re-use attacks and their mitigation

AUTHORS

JASON DAGIT
dagit@galois.com
Galois

SIMON WINWOOD
sjw@galois.com
Galois

GETTY RITTER
gdritter@galois.com
Galois

JEM BERKES
jberkes@galois.com
Galois

ADAM WICK
awick@galois.com
Galois

Code re-use attacks and their mitigation

July 18, 2017

Abstract

Code-Reuse Attacks (CRAs) are well studied in the academic community. In this article, we provide a brief summary of notable attacks and mitigations with a focus on Return-oriented Programming (ROP). Our goal is to provide a roadmap for readers who may or may not be familiar with CRAs and who want to become more familiar with the research. As this is a roadmap, our aim is to be broad and concise with executive summaries, including citations, but otherwise defer to the original publications for a detailed account.

We have included a glossary at the end of technical terms and acronyms. In addition, our bibliography includes more articles than are covered and we recommend the enthusiastic reader to review the bibliography to discover additional reading material in this area.

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-15-D-0035. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office.

Contents

1 Introduction

Before the broad availability of hardware-supported Data Execution Prevention (DEP), an attacker could simply take advantage of a program bug to inject malicious code, termed *shellcode*, into an application. For example, a traditional buffer overflow attack involved writing past the end of a stack-allocated array with a payload containing shellcode, along with a replacement return address pointing at the shellcode. Thus, when the function returned, the shellcode would be invoked, and the attacker would gain control of the program.

With the advent of DEP, attackers can no longer simply branch to injected shellcode. Rather, an attacker must now construct part of the attack from the code available, namely the program text of the application and its libraries. For example, an attack can inject shellcode as before, but now must arrange for the application itself to make the shellcode buffer executable and begin execution.

2 Attack Model

We assume an attacker can inject arbitrary data into some buffer in the application, and exploit some error in the program to force control flow to some desired location.

A CRA needs the following elements in order to succeed:

1. A bug or feature which allows the attacker to inject the exploit code;
2. A bug which allows the attacker to alter the control flow of the program;
3. A set of gadgets which
   (a) are sufficient to encode the operations desired by the attacker; and
   (b) are composable into the CRA program; and
4. The addresses of these gadgets.

Mitigating CRA attacks then involves removing at least one of these elements. We premise this report on the existence of the first two elements: bugs which allow an attacker to inject a CRA program, via, for example, a request URL; and a bug such as a buffer overflow or use-after-free which allows the attacker to force the program to branch to some location.

2.1 ROP concepts

In general, a *gadget* is a sequence of instructions which ends in an indirect control flow transfer, the target of which is controlled by the attacker. In practice, it is useful to distinguish *return* gadgets used in ROP [67], *call* gadgets used in Call-oriented Programming (COP) [17], and *jump* gadgets used in Jump-oriented Programming (JOP) [20]. A return, jump, or call gadget is an instruction sequence that ends in a return, indirect jump, or indirect call instruction, respectively. This distinction is useful as defenses will have different strategies to deal with the different classes. We will use the term CRA to encompass all code re-use attacks, disambiguating where necessary.

A gadget is typically chosen to perform one small operation, such as loading a register: such gadgets are easier to find and target in a ROP compiler. In general, however, a gadget may contain many instructions, some (or most) of which are benign, such as memory loads, conditional branches, and so forth. The challenge in using these gadgets is then to ensure that that any side effects will not cause exceptions (memory loads and stores are in valid memory, for example), and that conditional control flow takes the desired path.

A gadget may be part of an existing code block, or may be formed from part of another instruction: the x86 architecture has variable length instructions, and instructions may start at any address. Thus, an otherwise innocuous instruction may contain a useful unintentional gadget. For example, consider the following code block:

```
x  406b2e:       48 83 c4 58             add    $0x58,%rsp
x  406b32:       c3                      retq
```

The last byte of the first instruction is the hexadecimal value `0x58`, which corresponds to the opcode for moving the top of the stack into the register `rax`. Thus, the gadget starting at the address `0x406b31`

```
x  406b31:       58                      pop    %rax
x  406b32:       c3                      retq
```

contains the useful operation of popping of the rax register and returning.

A collection of gadgets is joined together to form a program. Such a program is constructed by carefully constructing a stack, for example, to link together gadgets through return instructions.

2.2 An example of a ROP attack

Perhaps the simplest ROP attack simply changes the permissions on a region of attacker controlled memory containing shellcode and then branches to a start address within that region. The implementation of this attack we shall use as an example is is to call the `mprotect` system call and then return to the shellcode.

Our attack then needs to copy into the `rdi` register the start of the region we wish to make executable, the `rsi` register amount of memory to change, and the `rdx` register flags indicating the new permissions, in this case the number 7 (for readable, writable, and executable.) We will make use of the following gadgets, all of which are obtained by branching into the middle of an instruction.

```
x  /* Load rdi */
x  4058b9:       5f                      pop    %rdi
x  4058ba:       c3                      retq

x  /* Load rsi */
x  407462:       5e                      pop    %rsi
x  407463:       c3                      retq

x  /* Load rdx */
x  40ade4:       5a                      pop    %rdx
x  40ade5:       48 98                   cltq
x  40ade7:       c3                      retq
```

The last gadget—the one which loads `rdx`—has a superfluous instruction; this instruction sign extends the `rax` register, and so can be safely included without unwanted side effects.

The program is then encoded as the following stack:

```
<address of shell code entry point>
<address of mprotect in glibc>
0x7
0x40ade4
<size of shell code region>
0x407462
<address of shell code region>     <- bottom of stack
```

When `mprotect` returns, execution will proceed from the shell code entry point.

We rely on some initial vulnerability to transfer control to address `0x4058b9` with the stack pointer pointing to the above stack. If we can overflow some stack allocated buffer, this may be as simple as overwriting the current stack with the above payload. A more complex vulnerability may require further gadgets to "pivot" the stack to an attacker controlled buffer.

One attack, for example, overwrites the virtual function table pointer such that invoking a method jumps to a gadget chain instead. In

```
x  41b87e:       48 8d 65 d8             lea    -0x28(%rbp),%rsp
x  41b882:       5b                      pop    %rbx
x  41b883:       41 5c                   pop    %r12
x  41b885:       41 5d                   pop    %r13
x  41b887:       41 5e                   pop    %r14
x  41b889:       41 5f                   pop    %r15
x  41b88b:       5d                      pop    %rbp
x  41b88c:       c3                      retq

x  41b9b1:       48 89 fd                mov    %rdi,%rbp     ; rdi :: *object.
x  41b9b4:       53                      push   %rbx
x  41b9b5:       48 89 f3                mov    %rsi,%rbx
x  41b9b8:       41 50                   push   %r8
x  41b9ba:       48 8b 3f                mov    (%rdi),%rdi   ; rdi :: **vtable
x  41b9bd:       48 85 ff                test   %rdi,%rdi
x  41b9c0:       74 06                   je     41b9c8
x  41b9c2:       48 8b 17                mov    (%rdi),%rdx   ; rdi :: *vtable
x  41b9c5:       ff 52 10                callq  *0x10(%rdx)
```

Currently, widely used ROP compilers, such as Mona[71] target hard-coded gadget chains: the tool includes a hand curated corpus of desirable gadgets which are used to form a ROP program. One notable exception to this approach is the Q tool [66] which uses techniques from program verification

to search for gadgets with particular behaviors. Although such techniques are not currently required, advances in applied ROP defenses may necessitate more complicated gadget discovery than a simple database.

3 Classes of Attacks

Here we draw attention to specific classes of CRA. Looking at each class allows us to form a better understanding of how the defenses may work. Furthermore, we may seek to prioritize addressing some classes over others due to technical factors such as feasibility of the mitigation or ubiquity of the attack.

3.1 Return-to-lib Attacks

We refer to attacks that reuse entire functions as *return-to-lib* [67] attacks. These attacks control an indirect branch and redirect it so that it branches to the natural start of an existing function and allows it to run normally. These attacks have been shown to be Turing Complete [75], but that is not necessary for a particular attack in this class to achieve its goals [41]. Often all the attacker needs is enough control to make a particular system call. For example, on POSIX the `mprotect` function can be used to make an attacker-controlled part of memory executable. Another example might include a return branch that is redirected to the `system` function call so that the attacker may execute a privileged shell.

This attack class is important because attacks in this class may look like normal function calls. For example, if the attacker controlled branch is an indirect call, then redirecting the branch to call a different function may appear indistinguishable from other function calls. Defenses against this sort of CRA need to do more than simply ensure function calls only happen at call sites.

Given a sufficiently granular representation of the application's call graph, it may be possible to determine that a call to an attacker-chosen function is not the intended behavior. To achieve such a policy, defenses generally need to know the valid set of destinations for each call site [32]. We refer to such defenses as *fine-grained* as contrasted against *coarse-grained* defenses.

Not only can return-to-lib attacks slip past coarse-grained defenses when they are initiated, functions called this way may also slip past fine-grained defenses when they return. For example, shadow stacks provide a fine grained defense for backward edges, but a typical shadow stack would permit these functions to return, as they were invoked using a call instruction.

3.2 Returnless ROP

As discussed in [20], it is not enough to simply protect `ret` instructions. The crux of the problem is that some sequences of instructions are equivalent to individual instructions. The simplest example of a sequence that is equivalent to a single `ret` instruction would be the sequence:

```
n+nfpop n+nv%rax
n+nfjmp n+nv%rax
```

The set of sequences that can be substituted is not simple to define. In particular, longer sequences can also be used, but will typically introduce additional side-conditions, side-effects, or constraints on their use. Depending on the context of the attack or the attacker's goals, these other sequences may be unsuitable, but it is hard to argue for a general case here.

Returnless kernels [54] are an example of a defense that attempts to defeat ROP by removing all `ret` instructions from the program (OS kernel), including unintentional instances of `ret` that occur as part of other sequences of instructions. This still leaves attack surface in the form of JOP, COP, `ret n`, and `iret` (although the last two are probably not present in practice). So while removing, or protecting, returns might make it harder to attack particular programs it does not give a full defense.

3.3 Counterfeit Object-Oriented Programming

In Counterfeit Object-Oriented Programming (COOP) [65] the authors show that C++ *vptrs* can be used to redirect control flow. The *vptrs* are controlled by creating "fake" objects. These fake objects have a memory layout that allows them to be compatible with the compiled C++ code. In the simplest form these fake objects have specially crafted vtables. The specially crafted vtables allow

the attacker to control dynamic dispatch. This way, a C++ program that invokes a virtual method on an object can be used to reuse arbitrary code in the address space.

The authors go on to refine the COOP approach to demonstrate that in the worst case it can be very subtle and hard to detect because injecting function pointers makes the attack easier but is not required.

3.4 COP & JOP

We refer to CRAs that focus on exploiting calls and jumps as COP and JOP, respectively. Additionally, we refer to both of these as forward edges in the control flow graph.

Many CRA mitigations focus on return instructions, but any indirect branch has the potential to be controlled by an attacker. A good example of a COP attack in practice is demonstrated by [34].

A shadow stack provides good protection for backward edges (e.g., returns) by tracking, at runtime, the legal addresses for returns. For backward edges, we know where functions should return to: the call site where they were invoked.[1] We might like to use analogous ideas to protect forward edges. Unfortunately, calculating the precise set of valid destinations of an indirect call or jump is undecidable [53] in the general case. If we use static analysis for this problem we are necessarily forced to use approximate answers.

Moreover, a COP attack using an indirect call instruction to make a function call will return in the normal way by passing a shadow stack, if present.

For these reasons, we consider protecting forward edges to be a harder problem than protecting backward edges.

3.5 Signal Return Oriented Programming

One of the more exotic ways to take control of indirect branching is an abuse of the POSIX [44] signal framework [14]. A Signal Return-Oriented Programming (SROP) attack works by putting a fake signal frame on the stack. The SROP attack triggers at the next invocation of the `sigreturn` system call. Signal frames contains a return address and in a normal signal frame this would tell the Operating System (OS) where to resume after the signal is handled. In the case of a fake signal frame, the attacker has carefully crafted the return address so as to execute a ROP gadget of their choosing.

Additionally, the OS does not track signals or enforce invariants regarding signal delivery. This allows the attacker to inject many fake signal frames and use `sigreturn` as a dispatch point for executing ROP gadgets.

One possible mitigation against SROP is for the OS to store the signal frame in protected memory instead of storing it in the process's memory. Another possible mitigation is for the kernel to insert a secret value into the signal frame, or compute a hash, to detect when the signal frame has been tampered with or fabricated. This would be analogous to stack canaries.

3.6 Just-In-Time Code Reuse

In [69] the authors observe that typical programs have multiple memory disclosure bugs. Snow *et al.* propose a technique for defeating some forms of Address Space Layout Randomization (ASLR). The ROP compiler they develop works as a Just-In-Time (JIT) compiler, meaning that it constructs the gadget set as part of the attack.

This ROP compiler works in phases and requires the attacker to find an appropriate memory disclosure exploit that can be adapted to fit the compiler's interface. The ROP compiler explores the program's memory disassembling any code it finds in memory and also looking for API and system calls. Locating API and system calls is crucial for interesting attacks, otherwise the attack cannot modify anything outside of its own address space. The ROP compiler is fairly advanced as it must allocate registers in sophisticated ways to take advantage of the gadgets it finds during dissasembly. Moreover, their example compiler is capable of syntax-directed translation given a Metasploit [50] script.

A notable constraint of this approach is that it requires the attacker to be able to execute some amount of code, the JIT ROP compiler itself, to bootstrap the attack. Given the availability of

[1] Some special support is needed for instances of exceptional control flow, such as `setjmp/longjmp`.

scripting environments built into common modern software, such as JavaScript or Flash in web browsers, this constraint may not be that big of a hurdle in practice.

4 Defenses

4.1 Evaluating Defenses

There are a large number of proposed defenses; in this section we discuss several criteria which are useful for evaluating CRA defenses.

Performance A defense which requires a significant performance degradation is unlikely to be adopted in a production system. The level at which a performance cost of an approach is considered significant is highly subjective; we can, however, examine security measures which have been implemented as a guideline. For example, the StackGuard [23] buffer-overflow defense has a reported overhead of approximately 7% (in 1998).

As a general rule, we will consider less than 1% performance overhead to be negligible.

Efficacy An *effective* defense is a defense which, broadly speaking, stops most or all CRA attacks. This metric is unfortunately very difficult to evaluate, as there is no well-founded scientific approach to showing the efficacy of a CRA defense. In the past, defenses which have been informally shown to be very strong, such as CCFIR [80], have later been successfully attacked [37].

Informal analysis of difficulty of circumvention is usually used as an argument for the soundness of CRA defenses, but it is possible for an approach to make attacks more difficult without completely eliminating CRA attacks.

Completeness & Applicability The *completeness* of a defense depends on how widely it can be applied throughout a software ecosystem: for example, a CRA defense that only worked on some programs would be less complete than one that worked on all possible programs. We should also consider to what programs a defense is *applicable*: for example, a defense may apply to arbitrary binaries, or only to newly recompiled source code, or only to source code that has particular programmer-or-tool-supplied annotation and analysis. Similarly, a defense may apply to all layers of the software stack, or may only apply to certain kinds of programs. Finally, a defense may run on stock hardware or with stock software, or might require specialized hardware or runtime software to accommodate the defense.

Resilience Some CRA defenses will become useless in the face of one violation, as the attacker can then reliably circumvent future checks or otherwise shirk future enforcement by the mechanism. Relatedly, some systems will only work if they are universally applied and will cease offering protection if other adjacent pieces of software are not protected by the same system. A defense which continues working in the face of previous violations, or in the face of incomplete application of the defense, is a *resilient* defense.

Measurement Challenges Evaluating a particular defense against the criteria outlined above can be difficult. For example, it is hard to measure the efficacy of a defense until an independent analysis is performed. Oftentimes a defense is proposed in the literature and later a paper is published that shows a weakness or exploit. The easiest criteria to measure is performance, but even that is hard to measure well.

Most defenses published in the literature use the Standard Performance Evaluation Corporation CPU benchmark suite (SPEC CPU) [39] to measure performance. Using an independently created and curated benchmark is good because it reduces bias in the experiment, increases reproducibility, and also makes it easier to compare different defenses. Unfortunately, SPEC CPU is not optimal for measuring the run-time impact of defenses on typical software. For instance, SPEC CPU does not include a web browser, or JavaScript interpreter, and most of the software included in the benchmark is designed around number-crunching. When benchmarking CRA defenses it is important to benchmark their effects on dynamic control flow, which SPEC CPU does not feature very heavily. This is not surprising, as the SPEC CPU benchmarks are designed to test compiler optimizations and architectural changes, but it does present a mismatch with our needs.

Completeness and applicability can be hard to measure without carrying an implementation through to deployment. Often times subtle interactions will surface when the defense is applied to a broader scope of programs, programming languages, and architectures.

4.2 Control-Flow Integrity

Control-Flow Integrity (CFI) was first introduced by Abadi *et al.* [5]. We use the term CFI to refer to not just to a particular mechanism of enforcement, but to the general policy or invariant that a program's control flow cannot be hijacked by an attacker.

The original defense outlined for ensuring CFI was based on inserting tags in the instruction stream of the program. In its most basic form, a check would be inserted at each dynamic branch site. The check would look at the tag at the destination site and compare it against a value hard-coded at the branch site. If the source and destination values were an exact match the branch was permitted, otherwise the program would be terminated.

The number of distinct tag values could be varied to achieve different CFI policies. For example, if one tag value is used throughout a program then any dynamic branch is permitted to end at any tagged location. At the other extreme, if we knew the precise set of legal destinations at each branch site we could use a different tag for each distinct set of destinations. We could also assign a distinct tag to jumps, calls, and returns, giving three tags. With this policy, any jump would be permitted to go to any valid jump destination, but not the destination of a call or return.

One example of an efficient coarse-grained policy appears in the NaCl sandboxing system [78], designed for executing untrusted native code within a web browser. NaCl performs static analysis on code before it is executed and disallows all dynamic jumps except for the two-instruction sequence referred to as "nacljump":

```
n+nfand n+nv%eaxp, l+m+mi0xfffffffe0
n+nfjmp p*n+nv%eax
```

This limits all dynamic jumps to a particular area of memory. However, the CFI policy in place here is designed only to prevent untrusted code from attacking the browser; a properly-designed payload could still perform an attack on the process running within the sandbox.

Fine-grained forward-edge control flow is often trickier to verify than backward-edge control flow, as backward-edge control flow can easily be ascertained by noting that control flow has returned to where it was before, whereas forward-edge control flow requires difficult and possibly undecidable [53] analyses of source code. One cheap approach is Virtual method Table Validation (VTV) [74], which specifically protects C++ virtual calls by performing run-time validation of the vtable pointer. This requires small source transformations to insert the validation code, but only needs easily-accessible type data in the C++ source to verify that the vtable pointer used to find a function location is within the set of allowable vtables for the given static class (i.e., that the vtable pointer belongs to that class or to a subclass of that class.)

4.3 Layout Randomization

CRAs often attempt to modify the call stack so that other specific library functions are called, so one trivial way of making CRAs more difficult to carry out is using ASLR. This technique will randomize the location of parts of the address space, making it unfeasible for an attacker to craft an exploit which can jump directly to a desired function. Various implementations of ASLR can be more fine-grained or coarse-grained depending on which locations are randomized, and can perform the randomization step at different times (e.g., at link-time or at run-time.)

Other work on layout randomization includes more fine-grained protection like Oxymoron [11], which uses a system not unlike the Procedure Linkage Table (PLT) or Global Offset Table (GOT) to introduce per-function indirection in a way that protects the translation table from any kind of observation. This produces strong forward-edge CFI but does not address backward-edge protection. Oxymoron was developed specifically to combat the capabilities of JIT ROP [69] (see Section 3.6), as JIT ROP is able to defeat existing fine-grained ASLR in the presence of memory disclosure bugs. Compact Control Flow Integrity and Randomization (CCFIR) also employs a variation of layout randomization, which is discussed in more detail in 4.6.

General forms of ASLR are not sufficient for fully protecting against CRAs, as attackers can still use various techniques to locate interesting parts of the address space [63].

4.4 G-Free

The G-Free approach [59] combines two different techniques for preventing CRAs, together implemented as a wrapper around the GNU Compiler Collection (GCC) and the GNU Assembler (GAS). The first is designed to remove unintentional gadgets by careful auditing and rewriting of the generated assembly. In this context, *unintentional gadgets* are sequences of bytes which were not intended to be valid executable code, e.g., fragments of adjacent instructions which can be interpreted as valid assembly if control flow were transferred to an unintended location. The second technique rewrites indirect control flow instructions with hand-written guard code that performs extra CFI verification before transferring control. Using this system, the authors were able to effectively remove all gadgets from the GNU C Library (glibc).

4.5 bin-CFI

The bin-CFI approach [81] takes an existing binary and produces a new one which contains a read-only version of the original binary as well as new executable code which implements the CFI protections. All branches in the rewritten code are redirected through a translation step to find the "actual" target followed by a trampoline that transfers control to the target. Other unusual control flow situations, such as Unix signals and dynamic linking through `ld.so`, require manual modification of relevant library functions in order to target the new executable code instead of the old binary. Some analysis of the target binary is performed in order to elucidate a control-flow graph, but by and large the actual CFI policy put into place by bin-CFI is a coarse-grained one.

4.6 CCFIR

In the CCFIR approach [80], each function call is replaced by a jump into a 'springboard' section. This section consists of call instructions arranged such that the return address is aligned to 16 bytes. By placing these springboard sections in regions with the 27th bit clear (i.e., every second 128M region) it is simple to check whether a return targets one of these springboard sections by checking bits in the return address, ensuring at least some backward-edge checking.

Indirect jumps/calls are handled similarly: there is an intermediate branch target in the springboard, one for each possible destination: at rewrite time, the pointers to functions are replaced by pointers to the stub in the springboard. Before a jump, the target is checked to ensure that it is a valid target (i.e., is in the springboard and has the correct alignment).

Finally, sensitive functions such as exec are noted and are not allowed to be returned to by non-sensitive functions, so that sensitive functions are unable to call regular functions.

This is all done with only limited binary analysis: instead, this system relies on the binary in question being relocatable, which means the binary includes enough information to allow all the roots of the Control Flow Graph (CFG) to be found by the rewriting tool. In practice, most modern applications are position-independent, so this restriction is not particularly onerous.

On the other hand, Göktaş *et al.* [37] show that it is possible to construct an attack on CCFIR: that is, they successfully construct a CRA payload which is allowed under the constraints presented by CCFIR.

4.7 Code Pointer Integrity

The Code Pointer Integrity (CPI) defense [52] partitions the address space into normal data and control-influencing or "sensitive" data. A "sensitive" object is one which contains a function pointer, or which contains a pointer to a sensitive object. The analysis of sensitivity is broad: C language features such as `void*` pointers and forward declarations are conservatively considered to be sensitive. Sensitivity is applied based on the types of objects in the source code, although more elaborate code analysis might produce a better sensitivity metric and therefore improve the overall performance of the system.

All access to sensitive objects is guarded by a layer of indirection which will ensure that the access is within the bounds of the object. The paper authors investigated various structures to use as maps from sensitive pointers to the actual objects referenced, and settled on a virtual linear array. The also build a "safe stack" which is accessed in much the same way as the sensitive data, except used for storing things like return addresses on the call stack.

4.8 Heuristic Defenses

The kBouncer [61] and ROPecker [22], defenses rely upon the observation that CRA programs, unlike normal execution, tend to be composed of chains of short instruction sequences linked by indirect branches. Thus, by checking whether the last n branches have less than some upper bound k instructions, a potential CRA attack can be detected.

The Last Branch Record (LBR) provides the required facility for recording these branches, and is used by both kBouncer and ROPecker. These approach differ in the manner of enforcement: kBouncer performs checks at system calls and, in the Windows system, at calls to sensitive API operations. The ROPecker approach, while guarding certain dangerous system calls (`mprotect` and `mmap2`), relies on virtual memory hardware to restrict the application to a limited set of executable pages. When the application branches outside this window, a CRA check is performed and the access is granted if successful. The performance of this approach relies on temporal and spatial locality of execution.

Heuristic-based approaches, while effective against programs produced, for example, by the Q compiler [66], are incomplete. Heuristic systems can be bypassed by carefully constructed attacks [17, 38]. For example, Göktaş *et al.* carefully constructed CRA program that can bypass these systems by ensuring that some gadget has a length greater than the threshold. In addition, these heuristics produce false positives for some applications. The authors suggest that per-application thresholds for program and gadget length may provide greater levels of security.

4.9 Hardware Defenses

As software-based CFI can have reasonably high overhead, various hardware-based mechanisms for CFI enforcement have also been explored.

Budiu *et al.* [16] propose a modification to the Alpha architecture that adds one `cfilabel` instruction which takes a 16-bit immediate value, and modifies three branch instructions to take a similar value. When a branch is executed, the immediate value at the branch and at the destination `cfilabel` must match. It is an error to branch to any instruction which is not a `cfilabel`. This is only a solution to forward-edge control flow; backward-edge control flow is protected by adopting, in addition to the hardware modifications, the hybrid software-based XFI [7] system.

In the Branch Regulation (BR) [48] system, calls are required to branch to the start of a function and function entry points are marked with a "br annotation". Returns are checked using a hardware-based shadow stack. The BR approach, instead of tagging direct jump targets like many other "landing-pad"-based hardware systems, requires that jumps merely target some location within the same function as the jump itself. This restriction ensures that jumps cannot be used to access unrelated code directly, but also severely interferes with common code structures (such as dynamically linked libraries) and optimizations (such as tail-call optimizations).

The HAFIX [10] system was designed for embedded hardware and involves four new CFI-enforcement instructions: `CFIBR`, `CFIDEL`, `CFIRET`, and `CFIREC`. The `CFIBR` instructions appears at the head of functions and, when executed, adds the current function to a set of active functions. The `CFIRET` instruction serves as a landing-pad instruction for returns, so returning to any other instruction is an error. Function exits are annotated with either `CFIDEL`—which directly removes the function in which it appears from the set of active functions—or `CFIRET`—which decrements an associated counter, thus allowing recursive functions to exit without removing themselves from the list of active functions until the associated counter reaches 0. Taken together, these ensure that returns can only be made to active functions. This system only addresses backward-edge control flow—no mechanism is put into place to ensure that calls are made to correct functions.

4.10 StackGuard

The idea behind StackGuard [23] is quite simple: it guards the return value on the stack by placing a "canary value" between the return address and the local variables. Then when a bug allows the program to write beyond the bounds of a local variable it also writes over the canary. The function epilogue code checks the canary value against a secret value. If the check passes it is assumed that the stack has not be tampered with. If the value does not match, then it assumed the return address is no longer valid and the program halts.

The primary weaknesses of StackGuard are (a) it must keep the canary value secret from the attacker, and (b) non-linear writes to the stack can skip over the canary value and corrupt the return

address. StackGuard is also only designed to protect return addresses on the stack. For example, it is not designed to protect against heap based attacks.

5 Implementation Concerns

5.1 Static Analysis

A fine-grained CFI policy needs a full dynamic call graph, which can be produced using points-to analysis of function pointers. It is easy to construct a CFG for static calls, but constructing a CFG for dynamic calls with arbitrary call-sites is undecidable in general [53]. Generally, call-graph construction techniques will over-approximate the possible call-graph to save time on the analysis, but a stronger CFI policy can be enforced by a more accurate call-graph, but at the cost of a slower initial analysis.

One additional problem with traditional control-flow analyses is that they typically require whole-program analysis. Many analyses face difficulties when used with dynamically-linked libraries. At best, these systems either provide a dramatic over-approximation of the call-graph when using external libraries, or require a computation of the complete call graph at dynamic link time.

It's possible to build a more conservative CFG by using actual execution traces as a source of control flow information. However, this suffers from an opposite problem: it's easy to *underapproximate* the set of valid destinations, especially for large programs with control flow that only happens in unusual cases. The result would be that a CFI enforcement mechanism might raise violations for perfectly fine code if that particular control flow sequence had not been observed in previous runs.

5.2 Dynamic Linking

Dynamic linking poses a problem for many CFI-enforcement mechanisms, as dynamic linking—by necessity—involves a complicated set of control flow transfers in order to access and execute external code in an on-demand way. For example, consider dynamic linking for ELF executables:

On Linux, an application consists of a number of *shared objects* including an executable object and zero or more shared libraries, such as the standard C library (libc). Each shared object may contain references to procedures or variables in other shared objects. Furthermore, multiple shared objects may define a single symbol—this allows, for example, applications to implement their own memory allocation by providing instances of `malloc` and `free`. The *dynamic linker* is responsible for determining how each symbol is resolved to a particular implementation.

Each shared object has a GOT which holds the address of objects which are not known at link time, including the addresses of function and objects imported from other modules. This table is constructed by the linker during the final link phase, and populated by the dynamic linker when the application is loaded.

Function calls from a shared object are performed through the PLT. The PLT is a per-shared-object set of procedures, each of which is responsible for invoking a function residing possibly, but not necessarily, in another shared object. The PLT finds the address of the actual implementation through a pointer stored in the GOT. Thus, a typical function call to a library involves calling into the corresponding PLT entry, which then branches to the address stored in the GOT.

The PLT serves a number of purposes; for example, it enables Position-Independent Code (PIC). Fundamentally, it allows the memory image for a shared object to be shared between multiple processes, potentially at different virtual addresses. Without the PLT, the dynamic linker would need to modify the text segment with the absolute address of each imported function, rendering that part of the shared object application-specific (and hence non-sharable).

In addition, the PLT allows for *lazy linking*, whereby the resolution of a reference is deferred until it is actually used: the corresponding GOT pointer initially points to a resolution routine in the dynamic linker. This resolution routine then overwrites the GOT entry with the address of the target procedure. Thus, after the first invocation, the PLT entry branches directly to the target procedure. Lazy linking is intended to improve the startup time of an application by omitting the resolution of symbols which may never be used, at the cost of slightly increased library invocation overhead.

All this necessarily means that the PLT builds a mechanism that is almost identical to a mechanism performed by an attacker: unusual control-flow that rewrites addresses to access particular library functions. Securing the PLT while preventing PLT-like capabilities from being accessible to attackers is a very difficult and involved task.

5.2.1 `dlopen()`

The `dlopen()` system call appears on most Unix-like operating systems, and allows a dynamic library to be included at run-time. This differs from typical dynamic linking in that a dynamically linked object will contain a table of all the symbols that it will eventually use, and can find those symbols at *link*-time, whereas `dlopen()` locates those symbols at *run*-time.

This poses a major problem for systems that want to protect against CRAs, because any use of `dlopen()` would necessarily involve dynamically extending the security policy to account for new code, which also means that the security policy must be mutable. This, in turn, means that the security policy might be open to modification by an attacker. Various mitigations may be possible—for example, means of verification of the symbols exported by `dlopen()` or static information about what is expected in the opened library—but a CRA-resistant `dlopen()` is an open and very complicated topic.

5.3 Kernel Enforcement

It is important that the operating system kernel be subject to security restrictions that are at least as stringent as the userland restrictions. Because the kernel has effectively unlimited access to the hardware and software of the machine, if the attacker is able to gain even a small amount of control over the kernel, the entire system could be compromised in drastic ways.

One way of building a kernel that is immune to similar attacks is by building a *returnless kernel* [54], which ensures at the compiler level that no `ret` instructions, will appear within the generated kernel source, including both intentional and unintentional `ret` instructions. All `call`/`ret` pairs are instead represented as control flow that passes through a fixed jump table. This was successfully applied to the FreeBSD operating system.

Some software-based mechanisms suffer from the possibility that a kernel-focused exploit, if successful, can simply disable the CFI-checking in some way, even if the CFI enforcement is otherwise resilient. Mandatory hardware checks are possibly stronger in this respect, as even a kernel exploit might not be able to remove the CFI protections from the running system.

5.4 Compiler limitations

The ideal policy for CFI has enough granularity to prevent hijacking of all indirect branches. As we have seen, a shadow stack can provide this for backward edges in the CFG. It accomplishes this by discovering the correct backward edge on the fly (i.e., from the previous call instruction.)

Ideally, we could have the same granularity of policy for forward edges. This turns out to be quite lot more challenging. The challenge lies in constructing the CFG. If we turn to static analysis to get the necessary information, we will soon run into practical and theoretical limitations. Due to well-studied [53] problems with aliasing and the semantics of C, and similar languages, it is intractable to do a good job of constructing the CFG statically. In general, constructing a precise CFG of the problem is undecidable.

The best we can hope to achieve is an approximation. If the approximation is conservative, meaning our approximation allows more destinations per-branch-site than the true CFG, then it presents an issue for security. If the approximate CFG is an under-approximation, then at run-time the program may make a legal branch that is flagged as illegal.

5.4.1 Loop unrolling

Loop unrolling is a common compiler optimization in which the contents of a loop are replicated multiple times in sequence. This—and similar optimizations—can alter the actual structure of the call graph, causing problems for some CRA defenses. Dynamic control flow sources or targets inside of the body of a loop being unrolled require careful attention.

5.5 Trade-offs

Real-world practical implementations face constraints that arise from for technical and historical reasons. In this section we list a small number of concerns that might arise in a real-world scenario, but these are by no means an exhaustive list of such concerns.

5.5.1 Performance vs. Completeness

One of the big challenges with designing a CFI enforcement mechanism is the trade off between overhead of the enforcement and the security gains of the enforcement. For example, a perfectly secure enforcement with 100% increase in runtime would generally be considered an unacceptable amount of overhead.

Probabilistic techniques, such as layout randomization, typically have lower overhead, but can be defeated completely if the attacker has access to the right information, such as with a memory disclosure bug.

Coarse grained CFI mechanisms intentionally make a trade-off in precision. This decrease in precision allows them to work for more programs and in more cases, but the research [17, 31, 38] has shown them to be overall weaker and defeatable in nearly, if not all, implementations.

A common strategy is to trade strictness of the CFI policy for faster or fewer tests. Davi *et al.* demonstrate that loosening the policy allows attacks to bypass the intended enforcement. One of the policies under consideration is a coarse-grained version of Abadi *et al.* CFI with relaxed indirect call checking. Furthermore, the authors break these defenses while assuming a policy that is the combination of several coarse-grained and heuristic defenses. This demonstrates that even using these defenses together is not sufficient to increase the security significantly.

5.5.2 Ubiquity & Systems support

Introducing CRA-protection into "the wild" will necessarily involve some amount of effort and replacement of existing infrastructure. A software-level mitigation will at least require recompilation of some software systems, and may require recompilation or even source-level modifications to wide swaths of existing code. A hardware-level modification will almost certainly require that all system code be in compliance with the mechanism.

It's possible to design a system that will support legacy code—for example, closed-source binaries which cannot be recompiled—but this will of course come at the cost of full-system security and would introduce weaknesses an attacker can target. It also brings up questions of how to mix legacy and protected modules, and whether some special mechanism is needed at their boundary.

Finally, a mechanism is unlikely to be adopted if it represents a significant break with the current code ecosystem. For example, a system which fails to support widely-used unusual control flow mechanisms, like C++ exceptions or threading systems, will likely not find adoption.

5.6 Proving Security

Perhaps the most important challenge is determining when the implementation of the CFI enforcement is actually correct. So far, most implementations rely either on informal proofs of correctness, or of other related heuristics: for example, many papers will count the number of possible gadgets in a program, and then count the number of gadgets following application of their CFI mechanism, and report the difference. This does speak to the fact that constructing a CRA against the system is more difficult; however, this by no means proves that an actual CRA would be impossible, or even *how* difficult a CRA would be in that context. In practice, CFI mechanisms have been demonstrated to be secure according to some metric, but later have been subject to various kinds of attacks. How to accurately and informatively show the degree of effectiveness of a CFI system is an open question.

5.7 Compliance

In the process of ensuring CFI for a program, many behaviors that are currently legal would become illegal. Some of these may be relied upon by the programs being compiled: for example, a JIT compiler may change the control-flow graph of the program at runtime in order to optimize program execution. How can a system detect whether legitimate programs become invalid by application of a CFI-enforcement mechanism, and how many legitimate programs are invalidated by these mechanisms? *Is* there a way of reliably detecting this?

6 Discussion

We have provided a survey of the major CRAs and mitigations currently available in the literature. We discuss metrics along which various CRA mitigations can be evaluated. Additionally, we have discussed various problems with those mitigations in particular as well as general concerns about how to effectively design and implement CRA mitigations in practical systems.

Glossary

ASLR Adress Space Layout Randomization, a system in which code is rearranged in a random way to prevent attackers from making assumptions about the layout of code in memory.

backward edge A jump or return to a previous location.

BR Branch Regulation, a CRA mitigation developed by Kayaalp *et al.* [48] by modifying the instruction set to expect particular control-flow-specific annotations within the source code.

buffer overflow A common vulnerability in which a program stores memory in a fixed-size buffer before validating that the memory is not too large to fit in the buffer, which can result in overwriting memory outside of the buffer.

canary A small randomly chosen value in a place which is likely to be overwritten in the event of a buffer overflow. This value can be checked afterward in order to observe whether a buffer overflow has occurred.

CCFIR Compact Control Flow Integrity and Randomization, a CRA mitigation developed by Zhang *et al.* [80] that relies on a "springboard" system to verify control flow.

CFG Control Flow Graph, the graph of possible jumps, calls, and returns that can happen within the execution of a program.

CFI Control-Flow Integrity, the property that a program's intended control flow cannot be altered by an attacker.

coarse-grained A CFI mechanism which ensures that the control-flow of a running program is broadly correct according to some metric but might still allow control transfers that are not present in the ideal CFG for the program. For example, the valid targets of a call instruction might be any valid function header, rather than a more specific set of functions tailored for that call instruction.

COOP Counterfeit Object-Oriented Programming, a technique in which an attacker overwrites the vtable pointers for objects to create dynamic calls to attacker-chosen functions.

COP Call-Oriented Programming, or building exploits by overwriting the target addresses of dynamic call instructions to point to different locations within existing binaries.

CPI Code Pointer Integrity, a CRA mitigation developed by Kuznetsov *et al.* [52] which puts "sensitive" data into a special section of the address space and requires that code accessing that data go through a layer of indirection.

CRA Code-Reuse Attack, a general name for the family of techniques in which an attacker modifies the state of a running program in such a way that the program executes existing code in the binary in an unintended way.

DEP Data Execution Prevention, the policy that allows chunks of memory to be *writeable* or *executable* but not both simultaneously.

fine-grained A CFI mechanism which ensures that the control-flow of a running program is as close to the ideal CFG as possible. For example, the valid targets of a call instruction would be restricted to exactly those functions which the programmer intended to be targets at that call site, and not any arbitrary function.

forward edge A jump or call to a new location.

gadget A sequence of instructions ending in an indirect control flow transfare that is usable as a component of a CRA attack.

GAS The GNU Assembler, an open-source assembler that forms the back-end to GCC.

GCC The GNU Compiler Collection, a popular open-source family of compilers.

glibc The GNU C Library, a standard open-source implementation of the C standard library.

GOT Global Offset Table, the table which contains the addresses of dynamically loaded symbols.

JIT Just-In-Time, used to describe compilers or other systems which produce assembly code at runtime.

JOP Jump-Oriented Programming, or building exploits by overwriting the target addresses of dynamic jump instructions to point to different locations within existing binaries.

LBR Last Branch Record, the registers in x86 processors that store the source and destination of the most recently executed branches.

loop unrolling A compiler optimization where the body of a loop is replicated inline several times with a corresponding decrease in the iteration count. This avoids the overhead of branching for each iteration of the loop.

OS Operating System, the software responsible for managing the hardware and facilitating the use of other software on a running computer.

PIC Position-Independent Code, code which makes no assumptions about where it is in memory and therefore can run regardless of its in-memory location.

PLT Procedure Linkage Table, the table used in dynamically linked binaries which contains "stub" functions that will locate and call the intended function.

returnless kernel A kernel which has been modified in such a way that the kernel source contains no `ret` instructions, either intentional or unintentional. Such a kernel is hardened against some kinds of CRA.

return-to-lib A CRA attack which reuses whole functions instead of smaller snippets of code.

ROP Return-Oriented Programming, or building exploits by overwriting return addresses to effect new control flow within existing binaries.

SPEC CPU The Standard Performance Evaluation Corporation CPU benchmark suite, a standard set of programs used to test compiler and procesor performance.

SROP Signal Return-Oriented Programming, a technique in which overwriting a signal stack frame gives the attacker the ability to jump to unintented locations in an existing binary after the signal returns.

unintentional gadget A gadget composed of instruction sequences found in bytes which were not intended to be interpreted as those instructions. For example, information in the payload of one instruction could be interpreted as an instruction on its own.

use-after-free A common vulnerability in which a reference still exists to an area of memory which had been marked as free and therefore might have been reused for other purposes. Using this "dangling reference" can affect memory in an unintentional way. This vulnerability can be used by a malicious attacker to take control of an executable.

vtable The table of function pointers which comprise the implementions of virtual methods for a C++ class. This is needed because virtual methods select an implementation at runtime and not at compile-time.

VTV Virtual method Table Validation, a CRA mitigation developed by Tice *et al.* [74] which performs runtime validation of the vtable pointer in C++ programs using class hierarchy information from the source program.

Bibliography

References

[1] The GNU compiler collection. http://gcc.gnu.org/onlinedocs/gccint/Plugins.html.

[2] The GNU C library. http://www.gnu.org/software/libc/libc.html.

[3] The linux kernel. http://kernel.org.

[4] The MySQL benchmark suite. http://dev.mysql.com/doc/refman/5.0/en/mysql-benchmarks.html.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Architectural support for software-based protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51. ACM, 2006.

[7] M. Abadi, M. Budiu, Ú. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, Nov. 2006.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[9] S. Andersen and V. Abella. Data execution prevention: Changes to functionality in microsoft windows XP service pack 2, part 3: Memory protection technologies". http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[10] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. Hafix: Hardware-assisted flow integrity extension. In *52nd Design Automation Conference (DAC)*, June 2015.

[11] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/backes.

[12] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.22. URL http://dx.doi.org/10.1109/SP.2014.22.

[14] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 243–258, May 2014. doi: 10.1109/SP.2014.23.

[15] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[16] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In J. Torrellas, editor, *ASID*, pages 42–51. ACM, 2006. ISBN 1-59593-576-2. URL http://dblp.uni-trier.de/db/conf/asplos/asid2006.html#BudiuEA06.

[17] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[18] M. Castro, M. Costa, and T. L. Harris. Securing software by enforcing data-flow integrity. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 147–160, 2006. URL http://www.usenix.org/events/osdi06/tech/castro.html.

[19] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, Inc., October 2009. URL http://research.microsoft.com/apps/pubs/default.aspx?id=101332.

[20] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866370. URL http://doi.acm.org/10.1145/1866307.1866370.

[21] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/chen.

[22] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. URL http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks.

[23] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267549.1267554.

[24] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, 2000. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=821514.

[25] 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 *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[26] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.26. URL http://dx.doi.org/10.1109/SP.2014.26.

[27] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*., April 2015.

[28] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[29] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A. reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS*, 2012.

[30] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[31] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi`.

[32] I. Diatchki, L. Pike, and L. Erkok. Practical considerations in control-flow integrity monitoring. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0: 537–544, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2011.27.

[33] U. Drepper. ELF handling for thread-local storage. `http://dev.gentoo.org/~dberkholz/articles/toolchain/tls.pdf`, 2003.

[34] C. Evans. The poisoned nul byte, 2014 edition. `http://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html`.

[35] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'15)*, May 2015.

[36] H. H. Feng, O. Kolesnikov, P. Fogla, and W. Gong. Anomaly detection using call stack information. In *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

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

[38] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas`.

[39] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34 (4):1–17, Sept. 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL `http://doi.acm.org/10.1145/1186736.1186737`.

[40] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4. doi: 10.1145/379605.379665. URL `http://doi.acm.org/10.1145/379605.379665`.

[41] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2012. USENIX. URL `https://www.usenix.org/conference/woot12/workshop-program/presentation/Homescu`.

[42] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, Jan. 1997. ISSN 0164-0925. doi: 10.1145/239912.239913. URL `http://doi.acm.org/10.1145/239912.239913`.

[43] Y. Huang, L. Peng, C. Wu, Y. Kashnikov, J. Rennecke, and G. Fursin. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*, Pisa, Italy, Jan. 2010. URL `http://hal.inria.fr/inria-00451106`. Google Summer of Code'09.

[44] IEEE and O. Group. Posix.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/.

[45] Intel. Intel 64 and IA-32 architectures software developer's manual, volume 2B: Instruction set reference A-Z. *Order Number 325383*, 2014.

[46] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Intel Corporation, 2014.

[47] R. Joiner, T. Reps, S. Jha, M. Dhawan, and V. Ganapathy. Efficient runtime-enforcement techniques for policy weaving. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 224–234, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635907. URL http://doi.acm.org/10.1145/2635868.2635907.

[48] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 94–105. IEEE, 2012.

[49] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362832.

[50] D. Kennedy, J. O'Gorman, D. Kearns, and M. Aharoni. *Metasploit: The Penetration Tester's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. ISBN 159327288X, 9781593272883.

[51] T. Kornau. Return oriented programming for the ARM architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[52] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://blogs.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[53] W. Landi et al. Undecidability of static analysis. *ACM letters on programming languages and systems*, 1(4):323–337, 1992.

[54] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755934. URL http://doi.acm.org/10.1145/1755913.1755934.

[55] B. Lynn. 64-bit linux return-oriented programming. http://crypto.stanford.edu/~blynn/rop/, 2012.

[56] M. Matz, J. Hubička, A. Jaeger, and M. Mitchel. System V application binary interface: Amd64 architecture processor supplement. Technical report, 2013. URL http://www.x86-64.org/documentation/abi.pdf.

[57] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 155–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1793-5. URL http://dl.acm.org/citation.cfm?id=827253.827734.

[58] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1317–1328, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660281. URL http://doi.acm.org/10.1145/2660267.2660281.

[59] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6. doi: 10.1145/1920261.1920269. URL http://doi.acm.org/10.1145/1920261.1920269.

[60] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[61] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas.

[62] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315260. URL http://doi.acm.org/10.1145/1315245.1315260.

[63] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 60–69, 2009. doi: 10.1109/ACSAC.2009.16. URL http://dx.doi.org/10.1109/ACSAC.2009.16.

[64] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 88–108. Springer International Publishing, 2014. ISBN 978-3-319-11378-4. doi: 10.1007/978-3-319-11379-1_5. URL http://dx.doi.org/10.1007/978-3-319-11379-1_5.

[65] 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 *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[66] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2028067.2028092.

[67] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL http://doi.acm.org/10.1145/1315245.1315313.

[68] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein. Systematic analysis of defenses against return-oriented programming. In S. Stolfo, A. Stavrou, and C. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41283-7. doi: 10.1007/978-3-642-41284-4_5. URL http://dx.doi.org/10.1007/978-3-642-41284-4_5.

[69] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.45. URL http://dx.doi.org/10.1109/SP.2013.45.

[70] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: 10.1145/237721.237727. URL http://doi.acm.org/10.1145/237721.237727.

[71] C. Team. The mona tool. https://github.com/corelan/mona.

[72] P. Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[73] The Santa Cruz Operation and AT&T. System V application binary interface. Technical report, 1997.

[74] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[75] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_7. URL http://dx.doi.org/10.1007/978-3-642-23644-0_7.

[76] F. Weimer. Cve-2014-6271: remote code execution through bash. 2014. URL http://seclists.org/oss-sec/2014/q3/650.

[77] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. Ripe: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076739. URL http://doi.acm.org/10.1145/2076732.2076739.

[78] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009. URL http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf.

[79] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046713. URL http://doi.acm.org/10.1145/2046707.2046713.

[80] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.44. URL http://dx.doi.org/10.1109/SP.2013.44.

[81] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.