The Continuing Arms Race: Code-Reuse Attacks and Defenses[1]

Stephen Crane, Immunant, Inc.
Andrei Homescu, Immunant, Inc.
Per Larsen, University of California, Irvine & Immunant, Inc.
Hamed Okhravi, MIT Lincoln Laboratory
Michael Franz, University of California, Irvine

[1] DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.
This material is based upon work partially supported by the Department of Defense under Defense Advanced Research Projects Agency (DARPA) contract FA8750-15-C-0124, Air Force contracts FA8721-05-C-0002 and FA8702-15-D-0001, and by the National Science Foundation under awards CNS-1513837 and CNS-1619211.
Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the Air Force, the National Science Foundation, or any other agency of the U.S. Government.

Contents

1 Diversity & Information Leaks **1**

1.1 Software Diversity 2

1.2 Information Leakage 2

1.3 Mitigating Information Leakage 3

1.4 Address Oblivious Code Reuse 7

1.5 Countering Address Oblivious Code Reuse 9

1.5.1 Authenticating Direct Calls & Returns 9

1.5.2 Securing **Indirect** Calls & Returns 11

1.6 Evaluation of Code Pointer Authentication 12

1.6.1 Security 12

1.6.2 Performance 14

1.7 Conclusions 15

List of Figures

1.1 Direct and indirect memory disclosure. Source Crane et al. [2015a]. 5

1.2 Code pointer authentication. Direct calls and returns are illustrated in the leftmost third of the figure; indirect calls and returns are shown in the rightmost two thirds. Light grey boxes contain execute-only code and white boxes contain data. Dark grey labels show where we insert additional instructions to prevent address harvesting attacks. The $=^{32}$ operator in the check after edge 9 indicates that we only check the lower 32-bits of the return cookie. 10

1.3 Performance overhead of code pointer authentication on SPEC CPU2006. All measurements include the overhead of the Readactor++ [Crane et al. 2015b] transformations. 14

List of Tables

1 Diversity & Information Leaks

Almost three decades ago, the Morris Worm infected thousands of UNIX workstations by, among other things, exploiting a buffer-overflow error in the *fingerd* daemon [Spafford 1989]. Buffer overflows are just one example of a larger class of memory (corruption) errors [Szekeres et al. 2013; van der Veen et al. 2012]. The root of the issue is that systems programming languages—C and its derivatives—expect programmers to access memory correctly and eschews runtime safety checks to maximize performance. There are three possible ways to address the security issues associated with memory corruption. One is to migrate away from these legacy languages which were designed four decades ago, long before computers were networked and thus exposed to remote adversaries. Another is to retrofit the legacy code with runtime safety checks. This is a great option whenever the, often substantial, cost of runtime checking is acceptable. In cases where legacy code must run at approximately the same speed, however, we must fall back to targeted mitigations which, unlike the other remedies, do not prevent memory corruption. Instead, mitigations make it harder, i.e., more labor intensive, to turn errors into exploits.

Since stack-based buffer overwrites were the basis of the first exploits, the first mitigations were focused on preventing the corresponding stack smashing exploits [Levy 1996]. The first mitigations worked by placing a canary, i.e., a random value checked before function returns, between the return address and any buffers that could overflow [Cowan et al. 1998]. Another countermeasure that is now ubiquitous makes the stack non-executable. Since then, numerous other countermeasures have appeared and the most efficient of those have made it into practice [Meer 2010]. While the common goal of countermeasures is to stop exploitation of memory corruption, their mechanisms differ widely. Generally speaking, countermeasures rely on randomization, enforcement, isolation, or a combination thereof. Address space layout randomization is the canonical example of a purely randomization-based technique. Control-flow integrity [Abadi et al. 2005; Burow et al. 2016] is a good example of a enforcement technique. Software-fault isolation, as the name implies, is a good example of an isolation scheme. Code-Pointer Integrity, CPI [Kuznetsov et al. 2014] is an isolation scheme focused on code pointers. While the rest of this chapter focuses on randomization-based mitigations, we stress that the best way to mitigate memory corruption vulnerabilities is to deploy multiple, different mitigation techniques as opposed to being overly reliant on any single defense.

1.1 Software Diversity

Randomization, or software diversity [Cohen 1993; Larsen et al. 2014b], essentially hides implementation details such as the memory layout from adversaries. This means that adversaries cannot rely on code, variables, or other program artifacts residing at a known location. This idea has similarities with biodiversity wherein some fraction of animals in a herd will have immunity against environmental hazards due to random differences in their immune systems. One can also draw parallels to kinetic warfare insofar that belligerents seek to conceal their locations to avoid becoming an easy target.

Because adversaries in the digital domain seek to exploit implementation flaws that trigger invalid memory accesses, the inputs that cause the unintended behavior are highly implementation dependent. This is why randomization of the code layout has a de-stabilizing effect on code-reuse attacks that depend on code snippets (gadgets in ROP parlance [Shacham 2007]) residing at known addresses.

Adversaries generally have two ways to bypass diversified binaries: guessing or reconnoitering their target. Repeatedly mounting an attack which crashes the victim program [Bittau et al. 2014; Evans et al. 2015; Shacham et al. 2004] has visible side effects that often facilitate detection. Information leakage, on the other hand, is often silent and leaves few traces, if any, on the victim system. In the rest of this chapter, we focus on bypasses of diversity relying on information leakage, particularly code layout disclosure, and the countermeasures available to defenders.

1.2 Information Leakage

In their seminal paper on stack guards, Cowan et al. mention that their techniques are not impossible to bypass, but to do so would require the attacker to examine the entire memory image of the program [Cowan et al. 1998, p. 4]. The tacit assumption is that the attacker cannot easily leak the memory contents of a running program. Their follow-up work focusing on pointers also cites the difficulty of accessing process memory in their security argument: "To obtain the key, the attacker would either have to already have permission to manipulate the process with debugging tools (e.g. ptrace) or would have to have already successfully perpetrated a buffer overflow attack against the process." [Cowan et al. 2003]. Strackx et al. [Strackx et al. 2009] were the first to examine what they termed the "Memory Secrecy Assumption" underpinning randomizing defenses at the time. The gist of their argument is that memory secrecy relies on the absence of memory corruption vulnerabilities, an assumption that, if valid would also obviate the need for memory corruption mitigations such as ASLR, stack canaries and other diversity techniques. Information leakage can arise from format string vulnerabilities which cause the defective program to print out internal data or code rather than the intended output. Strackx et al., point out that buffer over-reads are a more common source

of information leakage and demonstrate a concrete attack in which ASLR and ProPolice [Etoh and Yoda 2000] can be bypassed thanks to such over-reads.

Serna highlighted that type confusion and use-after-free vulnerabilities as well as application-specific vulnerabilities also facilitate information leakage [Serna 2012]. The talk also highlighted that the widespread deployment of ASLR and stack canaries in all modern operating systems had made information leakage a requirement to write reliable exploits. Most importantly, Serna noted that the combination of attacker-controlled scripting and memory corruption errors put adversaries in a powerful position.

Snow et al. [2013] translated Serna's observation into practice by using an overflowed buffer object to systematically scan the memory of the process running a malicious script. Just-in-time code reuse, JIT-ROP, attacks generalize previous attacks and are worth summarizing here. The general goal of JIT-ROP is to find as many mapped code pages as possible by starting from a small root set of known pages. The discovery of additional code pages happens by recursively scanning each page for references to other pages and adding these pages to a working set. In context of browsers, the JIT-ROP technique is used to break out of a sandboxed scripting environment, such a JavaScript VM hosted by a browser. This lets the adversary execute arbitrary code with all permissions granted to the operating system process. To do so, the adversary tricks an unsuspecting user into visiting a web page serving a malicious script. The script constructs a write-what-where primitive out of a memory corruption vulnerability such that the adversary can access any mapped location within the virtual address space of the process. Since the code layout is not known to the adversary a priori, the exploit fails if it touches unmapped memory and the resulting segmentation fault is not handled by the program. Segmentation faults are avoided by scanning for pointers to code in the data memory surrounding the overflowed object (using a priori knowledge of the heap layout). Next, the exploit scans the code page identified by the code pointer. Since the virtual-to-physical memory mapping happens at the page granularity, it is always safe to scan an entire page which is usually 4KiB in size. Snow et al. realized that they could implement a disassembler in JavaScript to recover references between code pages and use the recovered references to discover additional code pages recursively. The recursive disassembly step terminates when the script has discovered enough code snippets to mount a traditional code-reuse attack.

1.3 Mitigating Information Leakage

Backes and Nürnberger [2014] were first out of the gate with a response to JIT-ROP attacks. Their technique, Oxymoron, splits the code segment into 4KiB pages. Furthermore, any code reference to another page is indirected through a lookup table. The base of the lookup table is hidden using the vestiges of x86 segmentation. This prevents the recursive disassembly step in the JIT-ROP attack. An interesting aspect of Oxymoron is that the scheme was designed to allow code pages to be shared among processes. This is an important optimization for shared

libraries and one that is overlooked by most of the academic literature although it is crucial in practice.

Davi et al. [2015] presented a different response to JIT-ROP attacks—Isomeron—motivated by their finding that the original JIT-ROP technique could be modified slightly to bypass Oxymoron. The key to the Oxymoron bypass was the finding that data memory contains enough pointers to discover enough code pages to mount an attack, even if it is not possible to discover additional pages through inter-page references thanks to Oxymoron. Virtual method tables for the C++ dispatch mechanism, for example, enable pointer harvesting and lessen the need for recursive disassembly. The Isomeron defense [Davi et al. 2015] frustrates return-oriented programming techniques by cloning each program function and randomly picking between original and function clones during execution. Code-reuse exploits need not use returns to chain gadgets, so the Isomeron technique has shortcomings of its own.

Backes et al. [2014] advocated for a more principled way to counter information leakage: preventing read accesses to code pages. Their implementation—eXecute-no-Read or just XnR—presented a workaround for all x86 processors whose memory management units lack native support for executable, non-readable pages. To work around this limitation, XnR prevents reads by clearing the present bit for nearly all code pages. Normally, the CPU uses the present bit to track which pages are present in RAM and or paged out to disk. Accesses to a page with the present bit cleared, causes the CPU to generate a page fault which the operating system handles by reading the missing page from the pagefile. XnR modifies the operating systems page fault handler to to mark XnR pages present (without evicting their contents) if and only if the present bit was cleared to prevent read accesses *and* if the page fault was triggered by instruction fetch, i.e., an attempt to execute the page was made. If, on the other hand, the fault was generated by a read access to an executable page, the XnR page fault handler terminates the program before any memory contents can be leaked. The number of page faults to handle determines the overhead of the XnR approach. To avoid excessive slowdowns, XnR keeps a small window of recently executed pages readable and executable—and thus exposed to information leaks. However, XnR uses a sliding window of 2-8 pages to limit the amount of code that can be leaked at any point in the execution.

Gionta et al. [2015] developed a system—HideM—which similarly made code pages unreadable but does so by using the Translation Look-aside Buffer, TLB, in a special way known as TLB-desynchronization. On processors that use separate TLBs for data and code, the two TLBs are usually kept in sync which gives an executing process the same view of its address space regardless of the type of access. HideM configures the memory management unit such that accesses to the same virtual address translate to different physical addresses depending on the access type. This way, instruction fetches proceed as intended whereas read accesses—whether malicious or not—go to a different physical copies of the text section. To ensure that legitimate reads to constant data stored on code pages function correctly, HideM zeroes out all instructions in the readable copy of the text section while

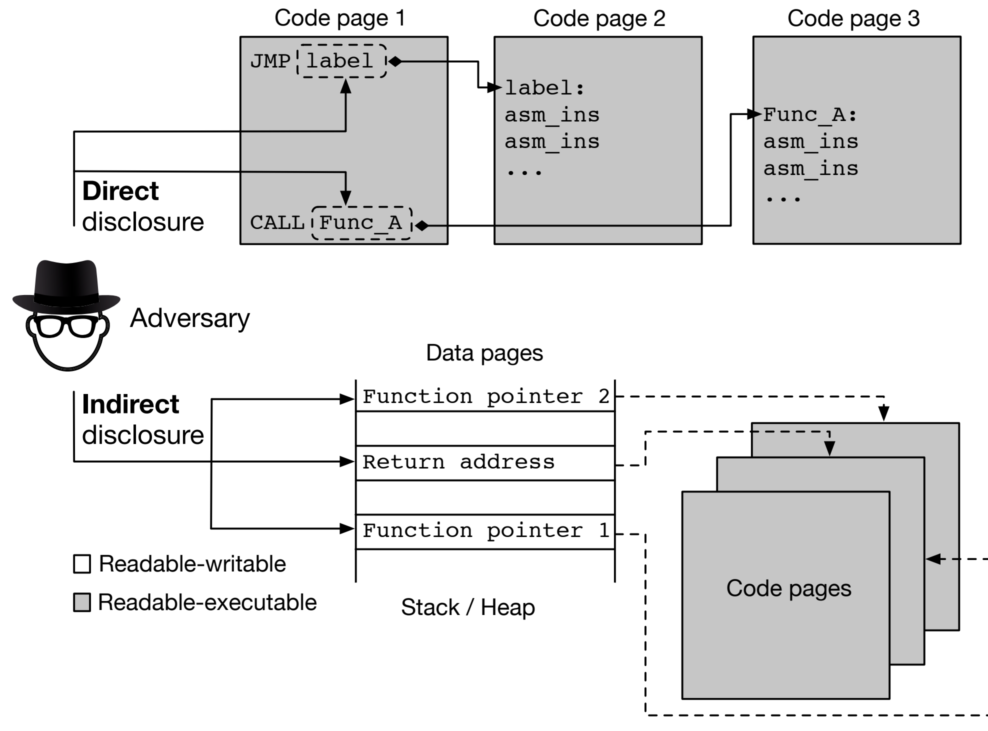

Figure 1.1 Direct and indirect memory disclosure. Source Crane et al. [2015a].

preserving all embedded constant data. This is a point in favor of HideM since XnR does not explicitly address the problem of reading embedded constants. On the other hand, most modern processors have unified TLBs and thus do not not support TLB-desynchronization as required by HideM.

While XnR and HideM goes a long way towards preventing *direct* leakage through adversarial reads, adversaries can also make inferences about the code layout by inspecting code pointers stored in the data segments of a running process. The difference between these two types of leakage is illustrated in Figure 1.1. The defenses we've discussed so far have protected the code pages and references between these (top half of Figure) but not references from data pages to code pages (bottom half of Figure). The utility of leaking a function pointer or return address when code pages cannot be read directly depends on the granularity of the code layout diversity. If each individual instruction is placed at a random location [Hiser et al. 2012], such leaks mainly facilitate whole-function reuse. However, the most granular diversity techniques tend to have high overheads [Larsen et al. 2014a] and may prevent page sharing between processes [Backes and Nürnberger 2014; Crane et al. 2016].

Crane et al. [2015a] built a system—Readactor—which explicitly seeks to prevent both direct and indirect leakage of code code layout. Rather than emulating execute-no-read permissions, Readactor leverages the extended page translation mechanisms found in modern (circa 2008 and onwards) processors to accelerate hypervisors. Memory accesses inside virtual machines undergo two levels of address translation i) guest virtual to guest physical translation and ii) guest physical to host physical translation. The effective permissions of an access to host physical memory is the intersection of the permissions used in the two translation steps. Unlike the first translation step which forces read permissions on executable pages, the second translation step can represent true execute-only memory permissions. The Readactor system used a lightweight hypervisor to activate the extended page tables on a per process basis to protect individual applications running on a traditional host system, i.e., outside a traditional hypervisor. Rather than allowing accesses to constant data on code pages, Redactor used a modified compiler to eliminate all such reads. The major open source C/C++ compilers later stopped emitting constants on code pages for performance reasons, which also benefits execute-only techniques.

Readactor tackles indirect leakage by introducing a pointer indirection layer so no pointer stored in a readable memory region points directly to its target. All that adversaries can observe are pointers into a special execute-only area containing trampolines (direct jumps) to the actual functions. Because the trampolines are stored on pages with execute-only memory, they cannot be dereferenced by an exploit. Adversaries therefore cannot learn the locations of functions in the absence of hardware-level side channels [Gras et al. 2017] or implementation errors. Readactor also demonstrated that just-in-time compiled code can be made compatible with execute-only memory with modest effort; the need to also protect JITed code from indirect disclosure was highlighted but not implemented. The necessity of avoiding indirect disclosure of JITed code was reiterated by Maisuradze et al. [2017].

A few variations of and extensions to the basic ideas behind XnR, HideM, and Readactor are worth mentioning. Schuster et al. [2015] demonstrated a new type of code-reuse attack called counterfeit object oriented programming, or COOP, which is capable of bypassing control-flow integrity defenses that are not *C++ aware*. C++-awareness, in this context, simply means using information about class hierarchies to further constrain the set of outgoing control flow edges at a C++ virtual method call site. C++-aware CFI is straightforward to implement when program source code is available whereas techniques to recover class hierarchies via binary analysis took a while to appear [Pawlowski et al. 2017]. Since COOP attacks execute entire C++ methods without regard for the actual code layout, such attacks can also bypass defenses such as Readactor. COOP attacks are not entirely layout agnostic however; they require knowledge of the layout of C++ objects and the layout of C++ virtual method tables. Since objects must be stored in RW memory, their layouts are difficult to hide. Vtables, on the other hand, contain a mix of data and pointers to code, the latter part of which can be hidden and randomized along the lines of the Readactor system. Crane et al. [2015b] presented a

counter to COOP attacks called Readactor++ that splits virtual method tables into two parts: one containing data and another containing code (direct jump trampolines to virtual methods). The code part, called the `xvtable`, is protected by execute-only permissions, and randomized. To prevent brute force attacks, dummy entries are added to each `xvtable` that are never activated during normal program execution [Crane et al. 2013].

Supporting execute-only memory is not always straightforward and most approaches rely on using the memory management unit in unconventional ways. For systems where MMU "tricks" are infeasible, such as systems having a simpler memory protection unit, execute-only permissions can be enforced in software [Braden et al. 2016] using techniques conceptually similar to software-fault isolation [McCamant and Morrisett 2006; Wahbe et al. 1993].

Lu et al. demonstrated that it is possible to use a pointer indirection layer to prevent indirect leakage *without* using execute-only memory to protect against direct leakage [Lu et al. 2015]. Their proposed solution, ASLR-Guard, uses the vestiges of x86 segmentation support to hide the location of a table that translates between code locators (visible to adversaries) and actual code addresses (hidden). Lu et al. argues that without a way to disclose code addresses, there is no need to prevent against direct leakage since a 64-bit virtual address space is large enough to resist brute force attempts at finding an ASLR'ed code segment. Later research on crash resistance and allocation oracles have undermined that assumption [Gawlik et al. 2016; Göktaş et al. 2016; Oikonomopoulos et al. 2016]. On a practical level, the ASLR-Guard implementation does not bound the growth of code locators, and thus its memory overhead.

Chen et al. [2017] demonstrated support for execute-only memory for source-less binaries. Specifically, their NORAX system is able to protect 64-bit ARM (AArch64) binaries. Notably, the AArch64 platform offers native support for execute-only memory unlike current x86 CPUs. A general challenge of binary analysis and assembly is to accurately separate code and data. Code misclassified as data (data misclassified as code) can lead to page faults when using DEP (execute-only memory) to mitigate exploits. NORAX addresses this challenge using a combination of offline binary rewriting and online load/runtime monitoring. The offline step conservatively estimates code regions and moves data bytes embedded in these regions to a new data section. The original data bytes are overridden with unique magic numbers that are recognized by the NORAX loader and runtime monitor. This lets the NORAX loader adjust any references to the original data bytes which are now inaccessible since all code is mapped with execute-only permissions. If an attempt to read a code page happens at runtime, the NORAX runtime monitor determines whether the associated access violation was generated by a legitimate access (missed by the offline analysis) or whether it is a malicious access which should cause program termination.

1.4 Address Oblivious Code Reuse

Rudd et al. [Rudd et al. 2017] explored the security properties of an ideal version of leakage-resilient code diversity, i.e., one that is not weakened by implementation-level flaws. Their

finding was that even an ideal implementation does not stop all types of code reuse. The reason is that code hiding mechanisms such as execute-only memory only apply to code pages, not code locators, e.g., function pointers and return addresses or pointers to Readactor trampolines. Code locators must be readable and writable for the program to function properly. Even with defenses such as Readactor and ASLR-Guard in place, adversaries can manipulate code locators used in place of traditional code pointers.

Rudd et al. used a data memory disclosure vulnerability to observe the state of a protected program as it executes. The fact that programs execute in a way that inherently leaks information about the state of execution enables profiling of the code indirection layer. An adversary can correlate the execution state of his own, unprotected program instance to that of a remote, protected instance at the time of the memory disclosure. Therefore, profiling can inform adversaries that a code identifier points to a function F in the protected program (without revealing the address of F). Adversaries can use this mapping from code identifiers to the underlying functions to construct a position-independent, whole-function code-reuse attack. Rudd et al. called this *address-oblivious code reuse*—AOCR—since the attack executes all code through code identifiers without any knowledge of the actual code layout.

Although AOCR attacks are possible, they require more effort to construct than their position-dependent equivalent. First of all, the state of the system changes rapidly which makes it challenging to correctly time memory disclosures of code identifiers. If the target application is multi-threaded, however, memory corruption allows an adversary to manipulate the variables controlling entry to a critical section. Mutexes, for instance, are usually set by a thread as it enters the mutex such that other threads wanting to enter will suspend until the first thread has exited the critical section protected by the mutex. For instance, an adversary may use one thread T_A to manipulate the mutex in a way that causes another thread T_B to block. This gives the adversary a chance to inspect memory without the timing unpredictability resulting from the execution of T_B.

Once the adversary has discovered a mapping from code locators to functions, he must find a way to i) hijack the control flow, ii) pass proper arguments to functions used in the exploit and iii) chain function calls. The control flow can be hijacked by using memory corruption to swap a code locator with the code locator corresponding to the first function in the malicious call chain. Rudd et al. solved the second challenge by reusing functions that read all their arguments from global variables. This requires knowledge of how global variables are laid out, but that too can be profiled and, in contrast to code, global variables must be readable. The third challenge, chaining calls through code locators, was solved using malicious loop redirection, MLR. This technique requires the vulnerable application to contain a loop whose body contains an indirect call site. Specifically, the loop must:

1. have a loop condition that is attacker controllable,
2. call functions through code pointers/locators.

An ideal loop looks like this:

```
while (task) { task->fptr(task->arg); task = task->next; }
```

where `task` points to a linked-list of (`fptr, arg`) pairs in attacker-controlled memory. Note that register randomization is not an effective defense because the semantics of the call dictates that the first argument is taken from `task->arg` and moved to `rdi` to conform to the x86_64 ABI. Note that MLR is conceptually similar to the loop-gadget concept in COOP and Subversive-C code-reuse attacks [Lettner et al. 2016; Schuster et al. 2015]

Using these techniques, Rudd et al. demonstrated working AOCR attacks against two popular web servers protected by Readactor: Nginx and the Apache HTTP Server. Readactor served as a stand in for leakage-resilient diversity techniques in general since it is the most comprehensive implementation of leakage-resilient diversity available. Note that approaches based on *destructive code reads* [Tang et al. 2015; Werner et al. 2016] are also vulnerable to AOCR since these attacks never attempt to read the actual code. Snow et al demonstrated additional attacks specifically targeting destructive-code-read techniques [Snow et al. 2016].

1.5 Countering Address Oblivious Code Reuse

Recall that code pointer hiding via trampolines already limits the set of addresses that are reachable from an attacker-controlled indirect branch. Even if an attacker discloses all trampoline pointers, only function entries, return sites, and individual instructions inside trampolines are exposed. We therefore implemented an extension to the Readactor code pointer hiding mechanism, which we call Code Pointer Authentication (CPA). CPA adds authentication after direct calls and before indirect calls to prevent the control-flow hijacking step as explained in Section 1.4 and thus mitigate AOCR attacks. One of the benefits of randomization-based defenses is that they do not rely on static program analysis, an advantage which helps them scale to complex code bases. To avoid relying on static program analysis, we must use different techniques to authenticate direct and indirect calls since we do not know the set of callees in advance.

1.5.1 Authenticating Direct Calls & Returns

Our general approach to authenticate direct calls uses cookies. A cookie is simply a randomly chosen value that is loaded into a register by the caller and read out and checked against an expected value by the callee. For returns, the callee loads another cookie into a register before returning, and the register is checked for the expected value directly after the return. Each function has two unique, random cookies: one to authenticate direct calls to the function (forward cookie, FC) and another to authenticate returns (return cookie, RC). Because the instructions that set and check cookies are stored in execute-only memory and the register storing the cookie is cleared directly after the check, attackers cannot leak or forge the cookies.

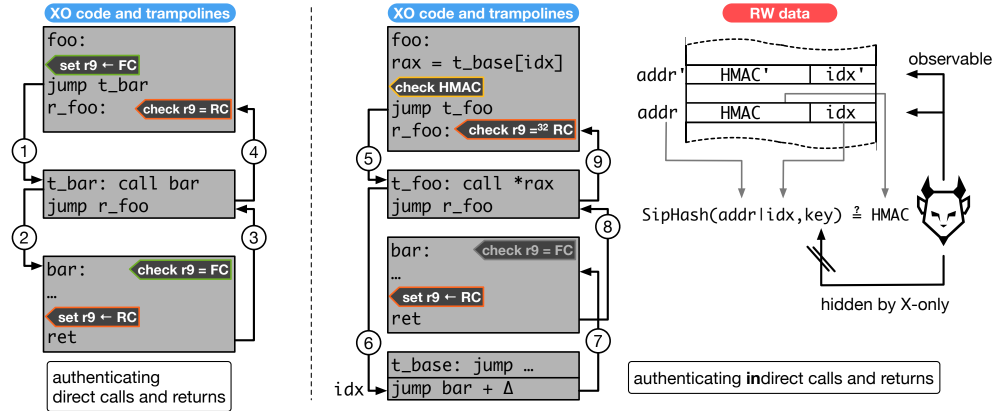

Figure 1.2 Code pointer authentication. Direct calls and returns are illustrated in the leftmost third of the figure; indirect calls and returns are shown in the rightmost two thirds. Light grey boxes contain execute-only code and white boxes contain data. Dark grey labels show where we insert additional instructions to prevent address harvesting attacks. The $=^{32}$ operator in the check after edge 9 indicates that we only check the lower 32-bits of the return cookie.

Our prototype implementation chooses cookie values at compile time and inserts these values into the execute-only code. A full-featured implementation could randomize the cookie values at load time so they vary between executions. This could easily be accomplished by marking all cookie locations during compilation, iterating over these locations during program initialization, and writing new cookies into the code before re-protecting the memory with execute-only permission.

The left-hand side of Figure 1.2 shows how we authenticate an example direct function call from `foo` to `bar`. Dark grey labels indicate how we extend the Readactor code pointer hiding technique with authentication cookies. Before transferring control to the direct call trampoline `t_bar` along control flow edge ①, we load `bar`'s forward cookie into a scratch register. Edge ② transfers control from `t_bar` to `bar`. The prologue of `bar` checks that the register contents match the expected forward cookie value and clears the register to prevent spilling its contents to memory. Before the `bar` function returns along edge ③, we load the backward cookie for bar into the same scratch register. At the return site in `foo`, we check that the register contains the backward cookie identifying `bar` as the callee. The return site then clears the register.

The return address pushed on the stack by the call instruction in `t_bar` leaks the location of the following jump instruction as well as the direct call itself. If the adversary manipulates an indirect branch to execute control flow edge ②, the check at the target address will cause the forward cookie check to fail and thus the attack to fail. Analogously, redirecting control to flow along edge ④ will cause the check at `r_foo` to fail.

1.5.2 Securing Indirect Calls & Returns

Without static program analysis, we don't know the target of an indirect call at compile time and thus enforce bounds on the program control flow. Cookies, as used in the direct call case, are therefore not applicable to indirect calls. However, we can still authenticate that the function pointer used in an indirect call was correctly stored and not maliciously forged without requiring any static analysis.

All function pointers in a program protected by Readactor are actually pointers to trampolines that obscure the true target address. Inspired by the techniques of CCFI [Mashtizadeh et al. 2015], we change the representation of trampoline pointers (which are stored in attacker observable memory) to allow for authentication. In Readactor's code pointer hiding mechanism, a trampoline pointer is simply the address of the forward trampoline. With CPA, the trampoline pointer representation is composed of a 16-bit index (`idx`) into a table of trampolines (starting at `t_base`) and a 48-bit hash-based message authentication code, HMAC. We show two such pointers in the right-hand side of Figure 1.2. Using a trampoline index prevents leakage of the forward trampoline pointer address since the base address of the array of forward trampolines `t_base` can be hidden in execute-only code. We found that programs need less than 2^{16} forward pointers in practice, so it suffices to use the lower 16 bits of a 64-bit word for the index (this can be adjusted as needed for larger applications). We compute the HMAC by hashing the index along with the least significant 48 bits of its virtual memory address. With this HMAC we can detect if the adversary tries to replace a code pointer with another pointer harvested from a different memory location. We find that SipHash [Aumasson and Bernstein 2012], which is optimized for short messages, is a good choice of HMAC for our approach.

The middle third of Figure 1.2 illustrates the case where the function `foo` calls `bar` indirectly through a function pointer. Again, dark gray labels highlight our extensions to Readactor's code pointer hiding technique. The indirect call site in `foo` loads the (HMAC, index) pair from memory, recomputes the HMAC using the (address, index, key) tuple, and compares the two (see rightmost third of Figure 1.2). If HMACs match, the index is used to lookup the address of the forward pointer which is subsequently used to execute control-flow edge ⑥. Note that the forward trampoline that creates edge ⑦ does not target the first instruction in `bar`; instead, we add a delta to the address of `bar` to skip the forward cookie check that authenticates direct calls to `bar` (e.g., edge ②).

As explained in Section 1.4, AOCR attacks swaps two pointers to hijack the program control flow. Because the address of the pointer is used to compute the HMAC, moving the pointer without re-computing the HMAC will cause the HMAC check before all indirect calls to fail unless the two (address, index) pairs collide in the hash. Attackers can still harvest and swap (HMAC, index) pairs stored to the same address at different times. See Section 1.6.1 for a more complete security analysis.

Returns from indirect calls make up the fourth and final class of control flows that we must authenticate. The callee sets a return cookie before the callee returns and checks the cookie at the return site; see edges ⑧ and ⑨ in Figure 1.2. We again clear the cookie register directly after the check to prevent leaks. The cookie check at the end of arrow ⑨ must pass for all potential callees. Therefore, we set the lower 32-bits of all backward cookies to the same global random value and only check the lower halfword of the backward cookie at the return site. This ensures that returns only target return sites; however, any return instruction can target indirect call-preceded gadgets under this scheme. We did not reuse any indirect call-preceded gadgets in our harvesting attack since these are also protected by register randomization and callee-saved stack slot randomization. It is possible to further restrict returns from indirect calls by taking function types into account. Rather than setting the 32 lower bits of return cookies to the same random value, we can use different random values for different types of functions.

1.6 Evaluation of Code Pointer Authentication

1.6.1 Security

Code pointer authentication prevents reuse of the remaining exposed trampoline pointers, even if the attacker has harvested all available trampoline locations. This authentication mitigates AOCR attacks. To show how, we systematically consider each possibly exposed indirect branch target in turn.

Direct call trampoline entry (edge ① in Figure 1.2) An attacker can harvest the location of the backwards jump (`jump r_foo`) in the call trampoline from the return address on the stack. In the original Readactor defense, it is possible to compute the address of the previous instruction from this pointer and invoke `t_bar`.

With direct call authentication, each direct callee function checks that its specific, per-function cookie is set prior to calling it. If the attacker cannot forge the callee function's cookie, this check will fail. We store the cookie as an immediate value in execute-only memory and pass it to the callee in a register. After performing the cookie check, the callee clears the register. Thus, direct call cookies cannot leak to an adversary, and the attacker has a 2^{-64} chance of successfully guessing the correct 64-bit random cookie value. Since the attacker cannot forge a correct cookie before an indirect branch to a direct call cookie, direct call trampoline entry points are unavailable as destinations for an attack.

Direct call trampoline return (edge ③ in Figure 1.2) Harvesting a return address corresponding to a direct call trampoline gives the attacker the location of the backwards jump in a call trampoline. In Readactor, this destination allows the attacker to invoke a call-preceded gadget beginning at `r_foo` in the example.

We also protect these destinations with an analogous, function-specific return cookie. Directly before a callee function returns, it sets its function-specific return cookie. The return site verifies that the expected callee's return cookie was set before continuing execution. This

prevents the attacker from reusing this destination unless the control-flow edge would be allowed during normal program execution.

Indirect call trampoline entry (edge ⑤ in Figure 1.2) Similarly, an attacker can harvest indirect call trampoline locations from the stack and dispatch to the beginning of an indirect call trampoline. However, this destination is trivial to the attacker, since he must set another valid, useful destination for the indirect call before invoking the trampoline. The attack could always dispatch straight to this final destination instead of the indirect call trampoline. Thus, we do not need to protect indirect call trampoline entry points from reuse.

Indirect call trampoline return (edge ⑧ in Figure 1.2) Analogous to the direct call case, the attacker can dispatch to the backwards edge of an indirect call trampoline to invoke an indirect-call proceeded gadget. This is a more challenging edge to protect without static analysis, since the indirect call site cannot know which function-specific return cookie to check.

Since the caller does not know the precise callee, we enforce a weaker authentication check on indirect call return destinations. By splitting return cookies into a global part and function-specific part, we can still ensure that the return site must be invoked by a return, not an indirect call. We believe that the fine-grained register randomization implemented in Readactor largely mitigates the threat of indirect-call proceeded gadget reuse, since the attacker cannot be sure of the semantics of the gadget due to execute-only memory.

Function trampolines (edge ⑥ in Figure 1.2) Function trampoline harvesting and reuse is the easiest attack vector against code-pointer hiding schemes. In Readactor, after harvesting function trampolines, the attacker can overwrite any return address or function pointer with a valid function trampoline destination and perform whole-function reuse.

We prevent reuse of function trampolines by changing the function pointer format to include an HMAC tying the function pointer to a specific memory address. This prevents reuse of function pointers from returns as well as most swaps of function pointers in memory.

Since function pointers are no longer memory addresses in our authentication scheme, the attacker cannot use a function pointer as a return address at all. The return would interpret the address as an HMAC—Idx pair and fail to verify the HMAC, crashing the program.

Function pointers cannot be swapped arbitrarily under this defense, since the pointer is tied to its address in memory by the HMAC. If a pointer P at address A is moved to address B, the HMAC check when it loaded from address B will fail. Thus the attacker must either forge a valid HMAC or have harvested P from the targeted location in memory at a previous point in execution.

HMAC Forgery We first address the possibility of forging a valid HMAC for a function and pointer address pair without ever having seen a valid HMAC for that pair. SipHash is designed to be forgery-resistant, thus the probability of correctly forging a valid HMAC for a pointer at an address not previously HMACed is expected to be 2^{-48}, based on the size of

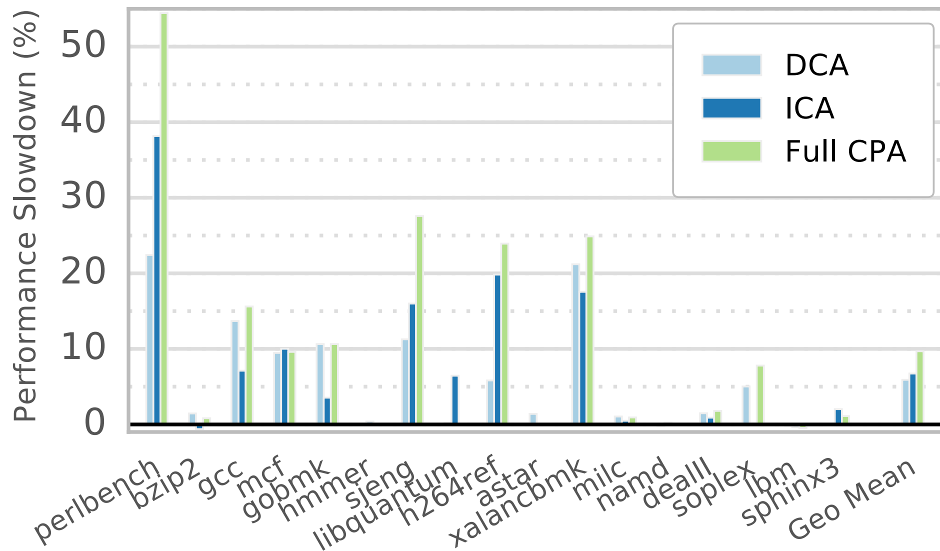

Figure 1.3 Performance overhead of code pointer authentication on SPEC CPU2006. All measurements include the overhead of the Readactor++ [Crane et al. 2015b] transformations.

the HMAC tag. Additionally, since we can store the HMAC key in execute-only memory, an attacker cannot disclose the 128-bit key, and thus is limited to brute-forcing this key.

Replay Attacks As in other pointer encryption schemes [Cowan et al. 2003; Mashtizadeh et al. 2015], HMACs do not provide temporal safety against replay attacks on function pointers. That is, a function pointer can be harvested at one point in program execution and later rewritten to the same address.

1.6.2 Performance

To evaluate the performance of our code pointer authentication, we applied the protections on top of the Readactor++ system. We measured the performance overhead of both direct call authentication and function pointer authentication on the SPEC CPU2006 benchmark suite. These results are summarized in Figure 1.3. All benchmarks were measured on a system with two Intel Xeon E5-2660 processors clocked at 2 Ghz running Ubuntu 14.04.

With all protections enabled, we measured a geometric mean performance overhead of 9.7%. This overhead includes the overhead from basic Readactor call and jump trampolines and compares favorably with the 6.4% average overhead reported by Crane et al. [Crane et al. 2015a]. We also measured the impact of direct call authentication and indirect call authentication individually (labeled DCA and ICA in the figure, respectively). We found

that indirect code pointer authentication generally adds more overhead (6.7% average) than direct code pointer authentication (5.9% average), although this is strongly influenced by the program workload, specifically the percentage of calls using function pointers.

We observed that h264ref stands out as an interesting outlier for indirect call authentication. This benchmark repeatedly makes a call through a function pointer in a hot loop. To make matters worse, the target function is a one-line getter, thus our instrumentation dominates the time spent in the callee. This benchmark in particular benefits greatly from inlining the HMAC verification to avoid the extra call overhead. To speed up HMAC verification, especially in this edge case, we implemented a small (128 byte), direct-mapped, hidden cache of valid HMAC entries. This hidden cache is only accessed via offsets embedded in execute-only memory and is thus tamper-resistant. Before recomputing an HMAC, the verification routine checks the cache to see if the HMAC is present.

We found three corner cases in SPEC where we could not automatically compute a new HMAC when a function pointer was moved. This is because the program first casts away the function pointer type then copies the pointer inside a struct. We had to insert a single manual HMAC in gcc and another in povray to handle these edge cases. perlbench stores function pointers in a growable list, which is moved during reallocation. Since our prototype does not yet instrument the libc `realloc` function, we had to manually instrument these operations. The CCFI [Mashtizadeh et al. 2015] HMAC scheme requires similar modifications. Finally, Readactor is not fully compatible with C++ exception handling, so we were not able to run omnetpp and povray which require exception handling.

1.7 Conclusions

There are three ways to bypass diversity-type mitigations. One is to target unprotected areas, the second is to employ brute force guessing, and the third relies on information leakage. The first two ways are relatively straightforward to counter through good engineering. The third option, however, remains the most challenging to fully address. Although it is possible to prevent leakage (perhaps modulo hardware side channels) leakage of the code layout, address oblivious attacks, though technically complex, are feasible. It is possible to mitigate address oblivious code reuse too, although the solution we designed and evaluated adds additional overhead and complexity to what was initially a fairly simple defense strategy.

If history is any guide, retrofitting security into fundamentally insecure languages without hampering performance will remain an open research challenge in the foreseeable future. The specific strand of research presented here is not the “one true answer” to all security problems; just as is the case with mitigation alternatives such as CFI and CPI. Instead, we describe our broader expectations for the short, medium, and long term based on recent industry developments:

- In the *short* term, deploying better mitigations is the best option. This is not a particular insight of ours; one simply has to look at the direction in which major software developers are headed. At the time of writing, work is underway to improve the granularity of code randomization schemes and hardware support for execute-only memory is forthcoming for Intel and already available for ARM. Although deployment of leakage-resilient diversity, as enabled by these techniques, is unlikely to stop all exploits, it does raise the bar to attackers considerably. At the same time, control-flow integrity techniques is supported by all major compilers and hardware support is similarly forthcoming from both Intel and ARM. Diversity and CFI are not mutually exclusive techniques; and either will stop a sufficiently determined adversary on its own. Rather, we believe a combination of disparate exploit mitigations will offer the best return on investment.
- Unlike the short term options, *medium* term options will require some source code changes. Access control mechanisms such as SELinux, when correctly implemented, helps implement the principle of least privilege such that vulnerabilities in unprivileged code cannot be used to carry out privileged operations. Legacy applications are unlikely to be broken into independent submodules based on the privileges they require, however. Therefore, manual refactoring may be required to realize the full potential of access control mechanisms. Similarly, techniques that retrofit type and memory safety into legacy C/C++ code require that bad casts and invalid memory accesses are removed from the application before a protected version can be released.
- Whereas medium term options may require minor changes and fixes to exiting source code, the best *long* term option is likely to very gradually retire C/C++ code. This will take multiple decades and some code bases may simply be abandoned instead as the software landscape changes anyway. The reason we mention language mitigation, however long it may take, is that it brings with it several important *secondary* benefits. Reduction of technical debt and the resulting productivity benefits are chief among these. C and its derivatives reflect the age in which they were designed. For instance, C programmers must declare variables and functions defined outside of the current translation unit such that the compiler can emit code in a single pass over the input files. Modern programming languages reflect the current reality that computing cycles are cheap and programmer attention scarce. Moreover, Balasubramanian et al. [Balasubramanian et al. 2017] showed that the features of the Rust systems programming language can support security capabilities, such as zero-copy software fault isolation, that cannot be implemented efficiently in traditional languages. Only by abandoning the languages in the C family, which have been spectacularly successful at any rate, can we make systems programming more productive, safe, and accessible.

Acknowledgments

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based upon work partially supported by the Department of Defense under Defense Advanced Research Projects Agency (DARPA) contract FA8750-15-C-0124, Air Force contracts FA8721-05-C-0002 and FA8702-15-D-0001, and by the National Science Foundation under awards CNS-1513837 and CNS-1619211.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the Air Force, the National Science Foundation, or any other agency of the U.S. Government.

Bibliography

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2005.

Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *13th International Conference on Cryptology in India*, INDOCRYPT, 2012.

Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security*, CCS, 2014.

Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, and Aurojit Panda. System programming in rust: Beyond safety. In *Workshop on Hot Topics in Operating Systems*, HotOS XVI, 2017.

Andrea Bittau, Adam Belay, Ali José Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, abs/1602.04056, 2016. URL http://arxiv.org/abs/1602.04056.

Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: enabling execute-only memory for COTS binaries on aarch64. In *IEEE Symposium on Security and Privacy*, S&P, 2017.

Frederick B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12, 1993.

Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, 1998.

Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, USENIX Sec, 2003.

Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby trapping software. In *New Security Paradigms Workshop*, NSPW, 2013.

Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015a.

Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM Conference on Computer and Communications Security*, CCS, 2015b.

Stephen Crane, Andrei Homescu, and Per Larsen. Code randomization: Haven't we solved this problem yet? In *IEEE Cybersecurity Development*, SecDev, 2016.

Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Division, Tokyo Research Laboratory, June 2000.

Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, CODASPY, 2015.

Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium*, 2016.

Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.

J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014a.

Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automatic software diversity. In *IEEE Symposium on Security and Privacy*, 2014b.

Julian Lettner, Benjamin Collenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. Subversive-c: Abusing and protecting dynamic message dispatch. In *USENIX Technical Conference*, ATC, 2016.

Elias Levy. Smashing the stack for fun and profit. *Phrack magazine*, 7, 1996.

Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

Giorgi Maisuradze, Michael Backes, and Christian Rossow. What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In *USENIX Security Symposium*, USENIX Sec, 2017.

Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

Stephen McCamant and Greg Morrisett. Sfi for a cisc architecture. In *15th USENIX Security Symposium*, USENIX Sec, 2006.

Haroon Meer. Memory corruption attacks: The (almost) complete history. In *Blackhat USA*, BH US, 2010.

Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hiearchies in ++ programs. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.

R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, Th. Hobson, S. Crane, Ch. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

Fermin J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012. URL https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf.

Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, CCS, 2007.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. of ACM CCS*, pages 298–307, 2004.

Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy*, 2016.

Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19 (1):17–57, January 1989. ISSN 0146-4833. doi: 10.1145/66093.66095. URL http://doi.acm.org/10.1145/66093.66095.

Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*,

EUROSEC, 2009.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, 2012.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating System Principles*, SOSP, 1993.

Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2016.

Author's Biography

Stephen Crane

To be added later

Andrei Homescu

To be added later

Per Larsen

Per is trying his hand as an entrepreneur and co-founded an information security startup—Immunant, Inc.—specializing in exploit mitigation. Previously, he worked four years as a postdoctoral scholar at the University of California, Irvine. He graduated with a PhD from the Technical University of Denmark in 2011.

Per co-organized the 2015 Dagstuhl Seminar upon which this book is based and has served as program committee member on several of academic conferences including USENIX Security, USENIX WOOT, ICDCS, and AsiaCCS. In 2015, he was recognized as a DARPA Riser.

Hamed Okhravi

Hamed Okhravi is a Senior Staff member at the Cyber Analytics and Decision Systems group of MIT Lincoln Laboratory, where he leads programs and conducts research in the area of systems security. His research interests include cyber security, science of security, security evaluation, and operating systems. He is the recipient of 2014 MIT Lincoln Laboratory Early Career Technical Achievement Award and 2015 Team Award for his work on cyber moving target research. He is also the recipient of an honorable mention (runner-up) at the 2015 NSA's 3rd Annual Best Scientific Cybersecurity Paper Competition. Currently, his research is focused on analyzing and developing system security defenses.

He has served as a program chair for the ACM CCS Moving Target Defense (MTD) workshop and program committee member for a number of academic conferences and workshops including ACM CCS, NDSS, RAID, AsiaCCS, ACNS, and IEEE SecDev.

Dr. Okhravi earned his MS and PhD in electrical and computer engineering from University of Illinois at Urbana-Champaign in 2006 and 2010, respectively.

Michael Franz

Michael Franz is the director of the Secure Systems and Software Laboratory at the University of California, Irvine (UCI). He is a Full Professor of Computer Science in UCI's Donald

Bren School of Information and Computer Sciences and a Full Professor of Electrical Engineering and Computer Science (by courtesy) in UCI's Henry Samueli School of Engineering. Prof. Franz was an early pioneer in the areas of mobile code and dynamic compilation. He created an early just-in-time compilation system, contributed to the theory and practice of continuous compilation and optimization, and co-invented the trace compilation technology that eventually became the JavaScript engine in Mozilla's Firefox browser. Franz received a Dr. sc. techn. degree in Computer Science and a Dipl. Informatik-Ing. ETH degree, both from the Swiss Federal Institute of Technology, ETH Zurich.