Opaque Control-Flow Integrity

Vishwath Mohan*, Per Larsen†, Stefan Brunthaler†, Kevin W. Hamlen*, and Michael Franz†
*{vishwath.mohan,hamlen}@utdallas.edu
The University of Texas at Dallas
†{perl,s.brunthaler,franz}@uci.edu
University of California, Irvine

***Abstract*—A new binary software randomization and Control-Flow Integrity (CFI) enforcement system is presented, which is the first to efficiently resist code-reuse attacks launched by informed adversaries who possess full knowledge of the in-memory code layout of victim programs. The defense mitigates a recent wave of *implementation disclosure attacks*, by which adversaries can exfiltrate in-memory code details in order to prepare code-reuse attacks (e.g., Return-Oriented Programming (ROP) attacks) that bypass fine-grained randomization defenses. Such implementation-aware attacks defeat traditional fine-grained randomization by undermining its assumption that the randomized locations of abusable code gadgets remain secret.**

Opaque CFI (O-CFI) overcomes this weakness through a novel combination of fine-grained code-randomization and coarse-grained control-flow integrity checking. It conceals the graph of hijackable control-flow edges even from attackers who can view the complete stack, heap, and binary code of the victim process. For maximal efficiency, the integrity checks are implemented using instructions that will soon be hardware-accelerated on commodity x86-x64 processors. The approach is highly practical since it does not require a modified compiler and can protect legacy binaries without access to source code. Experiments using our fully functional prototype implementation show that O-CFI provides significant probabilistic protection against ROP attacks launched by adversaries with complete code layout knowledge, and exhibits only 4.7% mean performance overhead on current hardware (with further overhead reductions to follow on forthcoming Intel processors).

I. Motivation

Code-reuse attacks (cf., [5]) have become a mainstay of software exploitation over the past several years, due to the rise of data execution protections that nullify traditional code-injection attacks. Rather than injecting malicious payload code directly onto the stack or heap, where modern data execution protections block it from being executed, attackers now ingeniously inject addresses of existing in-memory code fragments (*gadgets*) onto victim stacks, causing the victim process to execute its own binary code in an unanticipated order [38]. With a sufficiently large victim code section, the pool of exploitable gadgets becomes arbitrarily expressive (e.g., Turing-complete) [20], facilitating the construction of arbitrary attack payloads without the need for code-injection. Such payload construction has even been automated [34]. As a result, code-reuse has largely replaced code-injection as one of the top software security threats.

This has motivated copious work on defenses against code-reuse threats. Prior defenses can generally be categorized into: CFI [1] and artificial software diversity [8].

CFI restricts all of a program's runtime control-flows to a graph of whitelisted control-flow edges. Usually the graph is derived from the semantics of the program source code or a conservative disassembly of its binary code. As a result, CFI-protected programs reject control-flow hijacks that attempt to traverse edges not supported by the original program's semantics. *Fine-grained CFI* monitors indirect control-flows *precisely*; for example, function callees must return to their exact callers. Although such precision provides the highest security, it also tends to incur high performance overheads (e.g., 21% for precise caller-callee return-matching [1]). Because this overhead is often too high for industry adoption, researchers have proposed many optimized, coarser-grained variants of CFI. *Coarse-grained CFI* trades some security for better performance by reducing the precision of the checks. For example, functions must return to valid call *sites* (but not necessarily to the particular site that invoked the callee). Unfortunately, such relaxations have proved dangerous—a number of recent proof-of-concept exploits have shown how even minor relaxations of the control-flow policy can be exploited to effect attacks [6, 11, 18, 19]. Table I summarizes the impact of several of these recent exploits.

Artificial software diversity offers a different but complementary approach that randomizes programs in such a way that attacks succeeding against one program instance have a very low probability of success against other (independently randomized) instances of the same program. Probabilistic defenses rely on memory secrecy—i.e., the effects of randomization must remain hidden from attackers. One of the simplest and most widely adopted forms of artificial diversity is *Address Space Layout Randomization (ASLR)*, which randomizes the base addresses of program segments at load-time. Unfortunately, merely randomizing the base addresses does not yield sufficient entropy to preserve memory secrecy in many cases; there are numerous successful derandomization attacks against ASLR [13, 26, 36, 37, 39, 42]. Finer-grained diversity techniques obtain exponentially higher entropy by randomizing the relative distances between all code points. For example, binary-level *Self-Transforming Instruction Relocation* (STIR) [45] and compilers with randomized code-generation (e.g., [22]) have both realized fine-grained artificial diversity for production-level software at very low overheads.

Recently, a new wave of *implementation disclosure attacks* [4, 10, 35, 40] have threatened to undermine fine-grained artificial diversity defenses. Implementation disclosure attacks

NDSS '15, 8–11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23271

TABLE I. OVERVIEW OF CONTROL-FLOW INTEGRITY BYPASSES

		CFI [1]	bin-CFI [50]	CCFIR [49]	kBouncer [33]	ROPecker [7]	ROPGuard [16]	EMET [30]
DeMott [12]	Feb 2014							☹
Goktaş et al. [18]	May 2014	☹	☹	☹				
Davi et al. [11]	Aug 2014		☹		☹	☹	☹	☹
Goktaş et al. [19]	Aug 2014				☹	☹		
Carlini and Wagner [6]	Aug 2014				☹	☹		

exploit information leak vulnerabilities to read memory pages of victim processes at the discretion of the attacker. By reading the in-memory code sections, attackers violate the memory secrecy assumptions of artificial diversity, rendering their defenses ineffective. Since finding and closing all information leaks is well known to be prohibitively difficult and often intractable for many large software products, these attacks constitute a very dangerous development in the cyber-threat landscape; there is currently no well-established, practical defense.

This paper presents *Opaque CFI* (O-CFI): a new approach to coarse-grained CFI that strengthens fine-grained artificial diversity to withstand implementation disclosure attacks. The heart of O-CFI is a new form of control-flow check that conceals the graph of abusable control-flow edges even from attackers who have complete read-access to the randomized binary code, the stack, and the heap of victim processes. Such access only affords attackers knowledge of the intended (and therefore non-abusable) edges of the control-flow graph, not the edges left unprotected by the coarse-grained CFI implementation. Artificial diversification is employed to vary the set of unprotected edges between program instances, maintaining the probabilistic guarantees of fine-grained diversity.

Experiments show that O-CFI enjoys performance overheads comparable to standard fine-grained diversity and non-opaque, coarse-grained CFI. Moreover, O-CFI's control-flow checking logic is implemented using Intel x86/x64 memory-protection extensions (MPX) that are expected to be hardware-accelerated in commodity CPUs from 2015 onwards. We therefore expect even better performance for O-CFI in the near future.

Our contributions are as follows:

- We introduce O-CFI, the first low-overhead code-reuse defense that tolerates implementation disclosures.
- We describe our implementation of a fully functional prototype that protects stripped, x86 legacy binaries without source code.
- Analysis shows that O-CFI provides quantifiable security against state-of-the-art exploits—including JIT-ROP [40] and Blind-ROP [4].
- Performance evaluation yields competitive overheads of just 4.7% for computation-intensive programs.

II. THREAT MODEL

Our work is motivated by the emergence of attacks against fine-grained diversity and coarse-grained control-flow integrity. We therefore introduce these attacks and distill them into a single, unified threat model.

A. Bypassing Coarse-Grained CFI

Ideally, CFI permits only programmer-intended control-flow transfers during a program's execution. The typical approach is to assign a unique ID to each permissible indirect control-flow target, and check the IDs at runtime. Unfortunately, this introduces performance overhead proportional to the degree of the graph—the more overlaps between valid target sets of indirect branch instructions, the more IDs must be stored and checked at each branch. Moreover, perfect CFI cannot be realized with a purely static control-flow graph; for example, the permissible destinations of function returns depend on the calling context, which is only known at runtime. Fine-grained CFI therefore implements a dynamically computed shadow stack, incurring high overheads [1].

To avoid this, coarse-grained CFI implementations resort to a reduced-degree, static approximation of the control-flow graph, and merge identifiers at the cost of reduced security. For example, bin-CFI [49] and CCFIR [50] use at most three IDs per branch, and omit shadow stacks.

Recent work has demonstrated that these optimizations open exploitable security holes. By choosing ROP gadgets that start at a function entry point or are call-preceded, it is possible to build ROP chains that bypass CFI [19], including subverting CCFIR and bin-CFI. Related works [6, 11] have similarly shown that call-preceded gadgets can bypass bin-CFI as well as other low-overhead approaches that only check control-flow transfers before potentially dangerous function calls [7, 16, 30, 33]. Table I maps coarse-grained CFI approaches to the corresponding proof-of-concept bypasses. Note that the bypass of the original CFI approach assumes that returns are not tracked precisely using a shadow stack.

Just-In-Time Code Reuse. Until recently, most threat models for CFI and artificial diversity defenses assumed that the memory contents of protected processes were hidden from attackers. The advent of Just-In-Time ROP (JIT-ROP) [40] demonstrated that this assumption might be unrealistic in practice due to the existence of implementation disclosure vulnerabilities. Using heap feng shui [41], JIT-ROP places a buffer next to a string and a button object. By overflowing the buffer, the string length is set arbitrarily high, allowing the attacker to read any byte in the virtual address space. Parsing the button object through the overflowed string yields a reference to a mapped code page.

Typically, attackers need more than a single 4K page worth of code to find enough gadgets to mount a code-reuse attack. To discourage brute-force searches for more code pages, artificial diversity defenses routinely mine the address space with unmapped pages that abort the process if accessed [2]. JIT-ROP evades these mines by disassembling the initial code page and carefully traversing only direct references to other code pages to recursively discover enough gadgets to mount a ROP attack. Since gadget locations are no longer unknown to the attacker, reliable construction of custom ROP chains becomes possible despite the fine-grained randomization defense.

Blind ROP. While JIT-ROP targets scripting-enabled clients, Blind Return Oriented Programming (BROP) [4] targets vulnerable Internet-facing services, such as web-servers, that restart after a crash. It capitalizes on the observation that

child processes created with the `fork` system call on Linux must be randomized in the same way as their parent in order to continue executing. The attack uses a buffer overflow to overwrite the stack byte-by-byte. Byte values are chosen so that correct guesses cause the server to continue responding as intended, while incorrect guesses solicit a crash and restart. By distinguishing these two outcomes, attackers can remotely infer secret stack cookie values to bypass stack guards and discover gadget locations. Once the `write` system function is located (typically in less than 4000 guesses) the entire code section can be exfiltrated to an attacker-controlled server, after which a traditional ROP attack can be launched against the vulnerable system. Like JIT-ROP, the attack defeats ASLR, DEP, stack canaries and fine-grained code randomization on 64-bit systems.

Side Channel Disclosures. Recent work has even shown that under certain circumstances, gadget locations can be leaked through side channels, such as timing channels [35]. This underscores the difficulty of fully protecting software against all implementation disclosure vulnerabilities. Complete protection entails mitigation of all side channel information leaks, which is widely recognized as prohibitively difficult for most non-trivial software products.

B. Assumptions

Given these sobering realities, we adopt a conservative threat model that assumes that attackers will eventually find and disassemble all code pages in victim processes. Our threat model therefore assumes that the adversary knows the complete in-memory code layout—including the locations of any gadgets required to launch a ROP attack. We also assume that the attacker can read and write the full contents of the heap and stack, as well as any data structures used by the dynamic loader. In keeping with common practice, we assume that data execution protection is activated, so that code page permissions can be maintained as either writable or executable but not both.

However, we assume that attackers cannot safely perform a comprehensive, linear scan of virtual memory, since defenders may place unmapped guard pages at random locations. Instead, attackers must follow references from one disclosed memory page to another [40] or resort to guessing [4] in order to avoid inadvertently touching one of these mined pages and alerting defenders (e.g., triggering re-randomization). Successful attacks against our system are therefore those that reliably traverse control-flow edges not intended by the original program semantics without triggering an invalid access violation.

III. O-CFI Overview

O-CFI combines insights from CFI and automated software diversity. It extends CFI with a new, coarse-grained CFI enforcement strategy inspired by bounds-checking, that validates control-flow transfers without divulging the bounds against which their destinations are checked. Bounds-checking is fast, the bounds are easier to conceal than arbitrary gadget locations, and the bounds are randomizable. This imbues CFI and fine-grained software diversity with an additional layer of protection against code-reuse attacks aided by implementation disclosures. As a result, O-CFI enjoys performance similar to coarse-grained CFI, with probabilistic security guarantees similar to fine-grained artificial diversity in the absence of implementation disclosures.

Following traditional CFI, an O-CFI policy assigns to each indirect branch site a *destination set* that captures its set of permissible destination addresses. Such a graph can be derived from the program's source code or (with lesser precision) a conservative disassembly of its object code. We next reformulate this policy as a bounds-checking problem by reducing each destination set to only its minimal and maximal members. This policy approximation can be efficiently enforced by confining each branch to the memory-aligned addresses within its destination set range. All intended destination addresses are aligned within these bounds, so the enforcement conservatively preserves intended control-flows. Code layout is optimized to tighten the bounds, so that the set of unintended, aligned destinations within the bounds remains minimal. These few remaining unintended but reachable destinations are protected by the artificial diversity half of our approach.

Our artificial diversity approach probabilistically protects the aligned, in-bounds, but policy-violating control-flows by applying fine-grained randomization to the binary code at load-time. While the overall strategy for implementing this randomization step is based on prior works [45], its purpose in O-CFI is fundamentally different. Randomizing the code layout *does not conceal the new layout from attackers*, since our threat model assumes attackers can read all the randomized code. Rather, its purpose is to randomize the *bounds* to which each branch is constrained. The bounds imposed upon each branch are not disclosed by the binary code since bounds values are stored in protected memory, not expressed as immediate instruction arguments. Thus, attackers who can read the binary code must nevertheless guess which control-flow hijacks trigger an out-of-bounds branch violation and which do not.

Reformulating CFI in this way forces attackers to change their plan of attack. The recent attacks against coarse-grained CFI succeed by finding exploitable code that is reachable due to policy-relaxations needed for acceptable performance. These relaxations admit an alarming array of false-positives: instead of identifying the actual caller, all call-preceded instructions are incorrectly identified as permitted branch destinations. Such instructions saturate a typical address space, giving attackers too much wiggle room to build attacks. O-CFI counters this by changing the approximation approach: each branch destination is restricted to a relatively short span of aligned addresses, with all the bounds chosen pseudo-randomly at load-time. This greatly narrows the field of possible hijacks, and it removes the opportunity for attackers to analyze programs ahead of time for viable ROP gadget chains. In O-CFI, no two program instances admit the same set of ROP payloads, since the bounds are all randomized every time the program is loaded.

Since the security of coarse-grained CFI depends in part on the precision of its policy approximation, it is worthwhile to improve the precision by tightening the bounds imposed upon each branch. This effectively reduces the space of attacker guesses that might succeed in hijacking any given branch. To reduce this space as much as possible, we introduce a novel binary code optimization, called *portals*, that minimizes the distance covered by the lowest and greatest element of each indirect branch's destination set.

Our fine-grained artificial diversity implementation is an adaptation and extension of *binary stirring* [45]. Binary stirring randomizes the ordering of basic blocks within code sections each time a program binary is loaded into memory. The stirring has the effect of randomizing bounds to defeat attackers

TABLE II. PSEUDO-CODE TO CONSTRAIN BRANCH BOUNDS.

Description	Original code	Rewritten code

Indirect branches	`call/jmp` $\langle dest \rangle$	t := $\langle dest \rangle$
		t := t & $align\_mask$
		(b_{min}, b_{max}) := blt[$branch\_id$]
		if not $b_{min} \leq t \leq b_{max}$:
		  abort(#BR);
		call/jmp t

Returns	ret $\langle n \rangle$	[esp] := [esp] & $align\_mask$
		(b_{min}, b_{max}) := blt[$branch\_id$]
		if not $b_{min} \leq$ [esp] $\leq b_{max}$:
		  abort(#BR);
		ret $\langle n \rangle$

armed with implementation knowledge, and affords even higher probabilistic protections against attackers who lack such knowledge. Thus, O-CFI offers security that strictly subsumes and exceeds traditional fine-grained code randomization.

To protect against information leaks that might disclose bounds information, our implementation is carefully designed to keep all bounds opaque to external threats. They are randomly chosen at load-time (as a side-effect of binary stirring) and stored in a *bounds lookup table* (BLT) located at a randomly chosen base address. The table size is very small relative to the virtual address space, and attackers cannot safely perform brute-force scans of the full address space (see §II-B), so guessing the BLT's location is probabilistically infeasible for attackers. No code or data sections contain any pointer references to BLT addresses; all references are computed dynamically at load-time and stored henceforth exclusively in protected registers.

A. Bounding the Control Flow

For each indirect branch site with (non-empty) destination set D, O-CFI guards the branch instruction with a bounds-check that continues execution only if the impending target t satisfies $t \in [\min D, \max D]$. Indirect branch instructions include all control-flow transfer instructions that target computed destinations, including return instructions. Failure of the bounds-check solicits immediate process termination with an error code (for easier debugging). Termination could be replaced with a different intervention if desired, such as an automated attack analysis or alarm, followed by restart and re-randomization.

The bounds-check implementation first loads the pair $(\min D, \max D)$ from the BLT into registers via an indirect, indexed memory reference. The load instruction's arguments and syntax are independent of the BLT's location, concealing its address from attackers who can read the checking code. The impending branch target t is then checked against the loaded bounds. If the check succeeds, execution continues; otherwise the process immediately terminates with a bounds range (#BR) exception. The #BR exception helps distinguish between crashes and guessing attacks. To resist guessing attacks (e.g., BROP), web servers and other services should use this exception to trigger re-randomization as they restart.

Table II contains pseudocode for the guards. The BLT securely stores $(\min D, \max D)$ pairs for all branches, and is indexed using unique branch IDs ($branch\_id$ in Table II).

Following the approaches of PittSFIeld [29], NaCl [48], and Reins [46], O-CFI also aligns all policy-permitted indirect branch destinations to power-of-two addresses, and masks the low-order bits of all indirect branch arguments to force their targets to aligned addresses. This prevents attackers from diverting control to misaligned instructions that are not intended to be reachable by any legitimate flow of the original program. This is important since any gadgets formed from misaligned instructions do not receive bounds checks.

To bypass these checks, an attacker must craft a payload whose every gadget is properly aligned and falls within the bounds of the preceding gadget's conclusory indirect branch. The odds of guessing a reachable series of such gadgets decrease exponentially with the number of gadgets in the desired payload.

B. Opacifying Control-flow Bounds

Diversifying bounds. The bounds introduced by O-CFI constitute a coarse-grained CFI policy. Section II warns that such coarse granularity can lead to vulnerabilities. However, to exploit such vulnerabilities, attackers must discover which control-flows adhere to the CFI policy and which do not. To make the impermissible flows opaque to attackers, we use diversity. Our prototype uses a modified version of the technique outlined by Wartell et al. [45], which shuffles the basic block order at program load-time. The general approach could alternatively be implemented as a compiler-based defense for software whose source codes are available.

Performing fine-grained code randomization at load-time indirectly randomizes the ranges used to bound the control-flow. In contrast to other CFI techniques, attackers therefore do not have *a priori* knowledge of the control-flow bounds.

Preventing Information Leaks. Attackers bypass fine-grained diversity using information leaks, such as those described in §II-A. Were O-CFI's control-flow bounds expressed as constants in the instruction stream, attackers could bypass O-CFI via such leaks. To avoid this, we instead confine bounds information to an isolated data page, the BLT. The BLT is initialized at a random virtual address at load-time, and there are no pointer references (obfuscated or otherwise) to any BLT address in any code or data page in the process. This keeps it hidden from attackers.

We also take several additional steps to prevent accidental BLT disclosure via pointer leaks. Our prototype stores BLT base addresses in segment selectors—a legacy feature of all x86/x64 processors. Each load from the BLT indexes the `gs` selector to read the bounds. We only use `gs` in bounds checking instructions, so there are no other instructions that adversaries can reuse to learn its value. Attackers are also prevented from executing instructions that reveal segment register values, since such instructions are privileged.

To succeed, attackers must therefore (i) guess branch ranges, or (ii) guess the base address of the BLT. The odds of correctly guessing the location of the BLT are low enough to provide probabilistic protection. On 32-bit Windows systems, for instance, the chances of guessing the base address are

$$\frac{1}{2^{31}/2^{12}} = \frac{1}{524{,}288}$$

and on 64-bit Windows, the chances are

$$\frac{1}{2^{43}/2^{12}} = \frac{1}{2{,}147{,}483{,}648}$$

or less than one in two billion. Incorrect guesses alert defenders and trigger re-randomization with high probability (by accessing an unallocated memory page).

The likelihood of successfully guessing a reachable gadget chain is a function of the length of the chain and the span of the bounds. The next section therefore focuses on reducing the average bounds span.

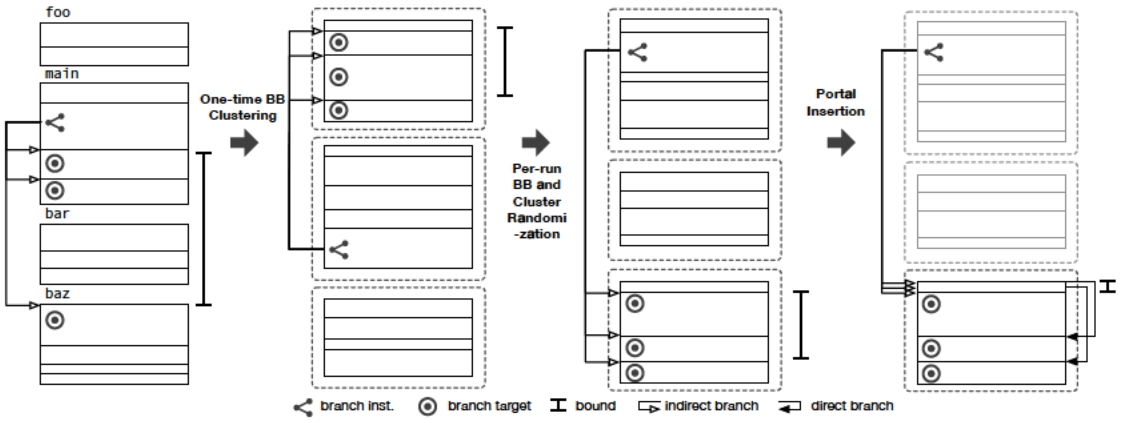

Fig. 1. O-CFI code layout transformation. Clustering occurs once, before the program executes (2nd column). Basic block and cluster randomization (3rd column), and portal insertion (4th column) occurs at load-time.

C. Tightening Control-flow Check Bounds

The distance between the lowest and highest intended destinations of any given indirect control-flow transfer instruction depends on the code layout. Placing indirect branches close to their targets both reduces bounds and improves locality, elevating both security and efficiency. Therefore we organize the code segment into *clusters*—one per indirect branch—each containing the basic blocks targeted by a particular branch.

To accommodate blocks that are destinations of multiple distinct branch instructions, we consider three options: (i) put the block in one cluster and expand the bounds of other branches to include its address, (ii) create duplicate copies of the block in multiple clusters, or (iii) add a *portal* block to each cluster, which unconditionally jumps to the block. Each solution incurs a trade-off: expanding bounds reduces security, creating duplicates increases code size, and portals introduce runtime overhead. The options are not mutually exclusive, affording optimizers a range of strategies. Our experiments indicate that portals are often the best choice (see below).

Figure 1 depicts the clustering, randomization, and portal creation phases, and their effects on the layout of the code segment—particularly with respect to bounds sizes. To prevent the load-time, fine-grained randomization phase from increasing the resulting bounds ranges (e.g., by placing two targets at opposite ends of the address space) and consequently nullifying the advantages of clustering, the load-time initializer proceeds in two stages, both depicted in the third column of Figure 1. First, it randomizes the locations of blocks within each cluster. This effectively changes instruction addresses without affecting the bounds of any of the indirect branches. Next, it randomizes the cluster order, further increasing entropy.

Minimizing Branch Ranges with Portals. A portal is a direct jump to a block. Jumping to a portal is therefore semantically equivalent to jumping directly to the block it targets. Placing a portal to a stray target within the cluster that contains all of the other targets of a branch avoids code duplication, and only widens the bounds by the portal size. Even when all n targets of a branch fall within a single cluster, bounds ranges can be further minimized by creating n portals to its n blocks. We call a collection of portals within a common cluster a *nexus*.

The *capacity* of the portal system limits the number of portals per nexus. Varying nexus capacity allows O-CFI to be tuned to different requirements. Setting it to zero prevents the creation of any portals, forcing the optimizer to choose alternative options. At the other extreme, setting no upper limit allows a portal to be created for *every* target, reducing all bounds ranges to wt, where w is the alignment width (usually 16 bytes; see §V-A) and t is the number of targets of the branch. At this setting, all indirect branches can only branch into a nexus, and through them, only to exactly those addresses that have been statically identified as targets. Thus, O-CFI with unbounded nexus capacity enforces fine-grained, static CFI.

The extra layer of indirection imposed by a portal has a minor impact on runtime; there is thus a trade-off between security and performance. Users may opt for full CFI enforcement with O-CFI for security-critical components, and lower the nexus capacity to a desired performance level for less critical software. In our experiments, we found that a nexus capacity of 12 results in a significant reduction in bounds sizes with imperceptible performance effects. All of our experiments in §V use this nexus capacity. Section V-D details how different nexus capacities affect bound ranges.

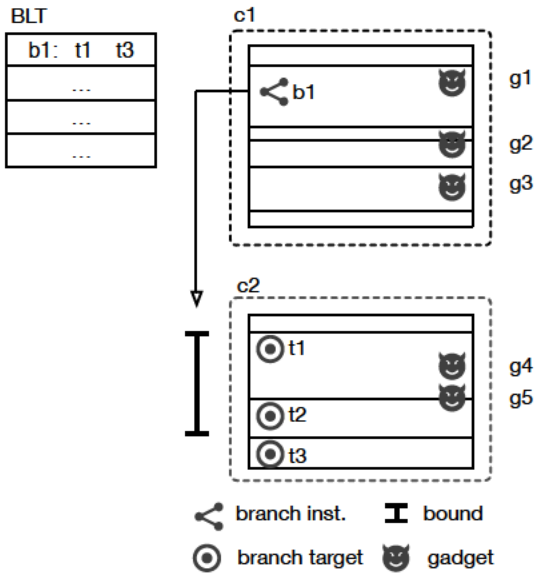

Fig. 2. Chaining gadgets in O-CFI

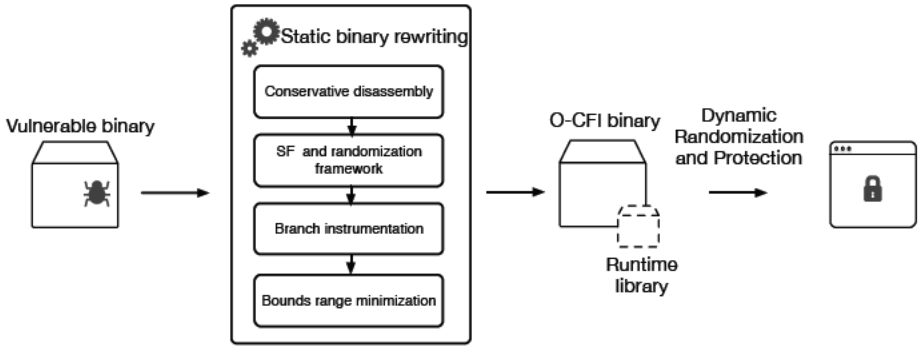

Fig. 3. O-CFI architecture. A vulnerable COTS x86 binary is analyzed and instrumented statically to create the corresponding O-CFI binary. At load-time, a runtime library added to the O-CFI binary randomizes the code layout and bounds the targets of every indirect branch.

D. Example Defense against JIT-ROP

The following example illustrates how O-CFI secures binaries against disclosure attacks. Consider a binary whose code segment contains five useful gadgets $g_1, \ldots, g_5$. Each gadget terminates in an indirect branch protected by a bounds check. Figure 2 shows such a binary after basic block clustering and randomization. Cluster c_2 contains the statically identified targets for the terminating indirect branch b_1 of gadget g_1. O-CFI constrains b_1 so it allows branches to a subset of the instructions contained in cluster c_2 (with similar constraints placed on other branches).

Under appropriate conditions, a disclosure attack such as JIT-ROP is able to recover a large portion of the runtime layout of the binary [40]. Our system does not prevent any of the six JIT-ROP steps. We allow code pages to be harvested, gadgets identified, and a payload to be dynamically crafted. We may even allow a few initial gadgets in the payload to execute. However, O-CFI hides the valid ranges for each gadgets from attackers even in the presence of implementation disclosure vulnerabilities. Recall that for this guessed payload to successfully execute, every gadget in the payload chain must fall within the valid range of the preceding gadget.

In our example, if g_1 is selected to be part of the payload, it can only be chained with gadget g_4 or g_5. Attempting to jump from g_1 to any other gadget triggers a bounds violation that stops the attack. Similarly, an attack that hijacks a control-flow to c_1 can only redirect it to gadgets g_1, g_2, or g_3; all other gadgets are outside cluster c_1 and are therefore detected as impermissible destinations of the hijacked branch. Broadly speaking, all links in a payload's chain must traverse edges in the Cartesian product of the (aligned) gadget sets within the corresponding clusters.

A successful attack must therefore limit itself to an extremely sparse graph of available edges. Our experiments (see §V-C) indicate that in practice the probability of successfully chaining gadgets in such a sparse graph is very low—just 0.01% for a four-gadget payload. The entropy of our procedure is further analyzed in §VI-A.

IV. O-CFI Implementation

We have implemented a fully functional prototype of O-CFI for the Intel x86 architecture. Our implementation uses a binary rewriting framework that secures COTS x86 binaries without source, debug, or relocation information. Like traditional CFI, however, we emphasize that O-CFI is equally suitable for inclusion in a compiler.

Our rewriter generates a transformed version of the binary that leverages

1) a coarse-grained CFI policy that bounds control-flows,
2) fine-grained randomization to thwart traditional ROP attacks and diversify control-flow bounds so they become unknown and unreliable for attackers,
3) x86 segmentation registers to prevent accidental leakage of the bounds lookup table (BLT), and
4) an SFI framework similar to PittSFIeld [29] to enforce instruction alignment, denying attackers access to misaligned instructions that bypass bounds checks.

The architecture of O-CFI is shown in Figure 3, and consists of two major phases. The first is a one-time static binary rewriting phase that minimizes bounds via clustering and inserts the runtime library that implements the dynamic phase. The second is a load-time, dynamic randomization and protection phase in which the runtime library randomizes the code layout, and populates the BLT to reflect the new locations of indirect branch targets. The dynamic phase also performs additional optimization, reducing bounds ranges to improve security.

A. Static Binary Rewriting

1) Conservative Disassembly: We first disassemble the code section using a conservative disassembler. Similar to the approach outlined by Wartell et al. [45], the code section is duplicated, with the old copy (renamed to `.told`) serving as a read-only data segment and the new copy (called `.tnew`) containing the rewritten executable code. The `.told` section is set non-executable, and all code blocks identified as possible targets of indirect jumps are overwritten with a five-byte tagged pointer. The tagged pointer consists of a tag byte (`0xF4`) followed by the four-byte address of that block in the `.tnew` section. The tag byte facilitates efficient runtime redirection of stale pointers to their correct targets, as explained below.

Since the `.told` section preserves all static data at its original addresses, data pointers in the rewritten code section continue to behave correctly. This makes the rewriting system resilient to disassembly errors that misclassify data as code.

Disassembly errors that misclassify code as data could omit such code from the `.tnew` section, resulting in a crash at runtime. To avoid this, we use settings that encourage the disassembler to interpret all bytes with valid instruction encodings as code. In our experiments, these settings suffice to avoid all disassembly errors that affect proper code translation.

TABLE III. MPX INSTRUCTIONS USED IN O-CFI

Syntax	Description
`bndmov bnd, m64`	Move upper and lower bound from `m64` to bound register `bnd`.
`bndcl bnd, r/m32`	Generate a #BR if `r/m32` is less than the lower bound in `bnd`.
`bndcu bnd, r/m32`	Generate a #BR is `r/m32` is higher than the upper bound in `bnd`.

2) SFI and Randomization Framework: To prevent attacks from jumping over the guards that constrain branch ranges, the new code segment is split into power-of-two sized basic-blocks called *chunks* [29]. Guard instructions and the branches they guard are always co-located within a common chunk, and branch targets are confined to chunk boundaries, with padding inserted where necessary to preserve this property. Confining branches to chunk boundaries is efficiently realized as a single `and` instruction per branch, which clears the lower i bits of the target address (where the size of a chunk is 2^{i+1} bytes).

Thus, control-flow within a chunk is linear, passing over every instruction from start to end. This chunking and masking regimen ensures that control-flow guards execute before every indirect branch. Additionally, it prevents jumps to misaligned (and hence unguarded) instruction sequences, reducing the attack surface to the set of statically disassembled (and hence protected) gadgets.

Direct branches are statically rewritten to reference their new target addresses. Indirect branches require extra effort, since their exact targets are only known at runtime. At runtime, there are two common cases: (a) the impending target is already within the `.tnew` section (e.g., it was pushed by a `call`), or (b) the impending target is a *stale pointer* that points into the `.told` section (e.g., it was loaded from a method dispatch table in the heap, which the static rewriter does not modify). The first case requires no special treatment; the second solicits an efficient dynamic lookup and redirection of the stale pointer to its new location [45]. Specifically, we check for the tag byte at the target address, and if present, substitute the current target with the address stored after the tag, which points to the block's new location in the `.tnew` section. The tag byte is chosen to be an illegal instruction encoding, so that no non-stale code pointer ever points to such a byte.

The stale pointer redirection mechanism is not relied upon for security. Like all indirect branch targets, redirected pointers undergo a mask and bounds-check before becoming control-flow destinations. Thus, corrupting or defeating the redirection mechanism does not circumvent the security policy.

The ability to redirect code pointers lays the foundation for load-time randomization. Once the new randomized locations for basic blocks have been finalized, updating the values in the `.told` section allows our redirection mechanism to correctly redirect all indirect branches to the new, randomized block locations. Direct branches are simply modified in-place.

3) Branch Instrumentation: The above techniques enforce SFI and fine-grained randomization. This protects against traditional ROP attacks, but not against implementation-aware attacks, which require the additional hardening implemented by O-CFI's bounds-checking. Bounds-checking is applied after stale pointer redirection alongside masking, to further limit the set of accessible gadgets. SFI enforcement prevents attack payloads from circumventing these bounds checks.

Algorithm 1 $CreateClusters(S)$: Cluster basic blocks to place the targets of indirect branches as close together as possible.

Input: S {the set of the basic blocks in the code segment}
Output: C {a set of clusters, one per indirect branch. Each $c \in C$ is a block set containing all targets of a specific branch, plus an empty nexus for later portal insertion.}

```
C ← ∅
for all b ∈ Branches(S) do
    c ← ∅
    for all t ∈ Targets(b) do
        b' ← GetBasicBlock(t)
        if b' ∉ ⋃C then c ← c ∪ {b'}
    end for
    {The nexus is an empty basic block to hold portals.}
    C ← C ∪ {(c ∪ CreateNexus())}
end for
{Add unclaimed basic-blocks into a single final cluster.}
C ← C ∪ {(S − ⋃C)}
```

Furthermore, due to randomization, the bounds remain unknown to implementation-aware attackers, and vary from program instance to program instance. Attacks cannot statically pre-compute bounds ranges because the runtime randomization phase changes bounds values on each execution. They also cannot dynamically leak the bounds, all of which are stored securely in the BLT and never leaked to the stack or heap. Attackers must therefore hazard guesses as to which gadget chains are safely accessible for any given program instance.

Our bounds-checking logic is detailed in Table II. The MPX implementation of this logic is assisted by the fast MPX instructions [24] summarized in Table III. On 32-bit systems, each BLT entry consists of two 32-bit pointers.

4) Accurate Target Identification: To ensure that we identify all intended targets of indirect branches, we employ disassembly heuristics that identify a superset of potential targets. As an example, we follow the following sequence of steps to identify the set of potential targets for a return instruction:

1) Identify all code references to the function that contains the return. This includes direct and indirect branches to the function entry point, as well as to any basic block within the function.
2) For each identified branch that is not a call, find all code references that flow into it.
3) Recursively traverse all non-call references until a fixed point is reached (i.e., a set with only calls).
4) The instruction immediately after each call forms the target set for that return.

Our heuristics are tuned to prefer false positives (non-targets treated as possibly valid destinations), since such errors do not significantly affect the operation of our system. In particular, each such error only marginally weakens the system's security (by admitting an unnecessary control-flow link that remains guarded by randomization) and slightly increases generated code size. A compiler-side solution could be more precise, at the cost of requiring source code and recompilation of programs.

5) Bounds Range Minimization: As discussed in §III-C, we use a combination of clustering and portals to reduce bounds ranges. While the portals themselves are created only at binary load-time, it is in the static phase that branch targets are clustered together and empty nexuses created. Algorithm 1 gives

TABLE IV. SUMMARY OF CODE TRANSFORMATIONS

Description	Original Code	Rewritten Code (MPX-mode)	Rewritten Code (Legacy-mode)

Indirect Branches	`call/jmp` $r/[m]$	`1: mov [esp-4], eax`
		`2: mov eax,` $r/[m]$
		`3: cmp byte ptr [eax], 0xF4`
		`4: cmovz eax, [eax+1]`
		— chunk boundary —
		`5: bndmov bnd1, gs:[`$branch\_id$`]`
		`6: bndcu bnd1, eax`
		`7: jmp 9`
		— chunk boundary —
		`8: xor eax, eax`
		`9: and al,` $align\_mask$
		`10: bndcl bnd1, eax`
		`11: xchg eax, [esp-4]`
		`12: call/jmp [esp-4]`
			`1: push ecx`
			`2: push eax`
			`3: mov eax,` $r/[m]$
			`4: cmp byte ptr [eax], 0xF4`
			`5: cmovz eax, [eax+1]`
			— chunk boundary —
			`6: mov ecx,` $branch\_id$
			`7: cmp eax, gs:[ecx]`
			`8: jb 10`
			`9: cmp gs:[ecx+4], eax`
			`10: jbe abort` (boundary — chunk)
			`11: and al,` $align\_mask$
			`12: xchg eax, [esp]`
			`13: pop ecx`
			`14: pop ecx`
			`15: call/jmp [esp-8]`

Returns	`ret` $\langle n \rangle$	— chunk boundary —
		`1: xchg eax, [esp]`
		`2: and al,` $align\_mask$
		`3: bndmov bnd1, gs:[`$branch\_id$`]`
		`4: jmp 6`
		— chunk boundary —
		`5: xor eax, eax`
		`6: bndcu bnd1, eax`
		`7: bndcl bnd1, eax`
		`8: xchg eax, [esp]`
		`9: ret` $\langle n \rangle$
			— chunk boundary —
			`1: xchg eax, [esp]`
			`2: cmp eax, gs:[`$branch\_id$`]`
			`3: jb 9`
			`4: and al,` $align\_mask$ (boundary — chunk)
			`5: cmp eax, gs:[`$branch\_id + 4$`]`
			`6: jae 9`
			`7: xchg eax, [esp]`
			`8: ret` $\langle n \rangle$
			— chunk boundary —
			`9: jmp abort`

a high level overview of our clustering algorithm. Each cluster created in this step gets an empty nexus. In our implementation, all nexuses are homogeneous in size, but more sophisticated implementations could tailor nexus sizes to individual branches based on the size of their statically determined target set.

Organizing the code into power-of-two sized chunks (for SFI enforcement) impacts portals. In the absence of chunking, the size of each portal is the five bytes required for a direct jump; but chunking rounds this up to the nearest multiple of the chunk-size. In our implementation, this makes each portal 16 bytes long. Though this slightly increases both file and code sizes, it only marginally affects the average bounds size, and does not noticeably impact performance. Section V-D contains a detailed breakdown of how bounds sizes vary with the number of portals per cluster.

B. Accelerated Bounds Checks

To optimize performance, we leverage the Intel memory-protection extensions (MPX) for x86/64 architectures to store and check bounds. MPX instructions will be supported in Intel processors from 2015 onwards, so our approach will benefit from hardware acceleration in the near future. MPX instructions execute as NOPs on legacy processors.

MPX provides hardware-accelerated bounds checking instructions and registers, for protection against buffer overflow or underflow attacks. The eight new bounds registers each hold two pointers, and can be used to store both the lower and upper bounds associated with a pointer value. New MPX instructions allow for quick loading and testing of these bounds registers. We use the three MPX instructions shown in Table III. Instruction `bndmov` loads bounds from the BLT into bounds register `bnd`, and `bndcl` and `bndcu` verify that the target address is within the loaded bounds.

To secure binaries intended for use on non-MPX systems, O-CFI also has a legacy mode that uses the `cmp` and `jcc` instructions to guard branches. Although the lack of dedicated range checking instructions makes these guards less efficient than their MPX-enabled counterparts, binaries rewritten in this mode receive the same level of protection.

Table IV shows the final consolidated sequence of instructions that enforces bounds, prevents execution of unintended instructions, and allows fine-grained randomization. Column three shows the instructions used when targeting MPX-compatible platforms, while column four shows those used on non-MPX, legacy processors. Chunks are 16 bytes each. In each listing, instructions appearing before the first chunk boundary are appended to the preceding chunk, or wherever they best fit. Subsequent instructions are confined to dedicated chunks in order to maintain security.

MPX Mode. In MPX mode, lines 1 and 11 of the guard code for indirect branches preserve the `eax` register, which is used as a scratch space. Lines 3 and 4 implement the dynamic lookup and redirection mechanism for stale code pointers. Lines 5, 6, and 10 load the bounds associated with this branch into bounds register `bnd1` and then compare it against the target address. If the target address is outside the bounds, a #BR exception is raised, and the program halted. Line 9 masks the target address, forcing it to a chunk boundary.

Line 8 foils hijackers who attempt to abuse the final chunk as a gadget. The earlier chunk boundary (above line 5) needs no such protection because all logic above it is strictly for preserving program functionality, not for enforcing security. Thus, jumping to that boundary during a code-reuse attack does not help the attacker—the resulting gadget implements a fully guarded jump.

The process is shorter for returns, since returns do not require stale pointer correction. The full return guard code therefore fits within two 16-byte chunks.

Legacy Mode. To protect binaries executing on processors without MPX support, O-CFI emits legacy mode guards. This mode uses comparison (`cmp`) instructions (lines 7 and 9 of the

Algorithm 2 $RuntimeSetup(C, BLT)$: Perform runtime randomization, and bounds range setup and optimization.

Input: C {clustered code segment}, BLT {bounds table}
Output: C {randomized, bounds optimized code segment}
  $RandomizeCode(C)$
  $CreateAllPortals(C)$
  $UpdateDirectBranches(C)$
  $UpdateJumpTable(C)$
  $UpdateBoundsTable(BLT)$
  $SetupSegmentedAccess(BLT)$ {Move the bounds table to a random page and set up segmented memory access to it via the `gs` register.}

Algorithm 3 $RandomizeCode(C, Shuffle)$: Randomize basic blocks in a cluster-aware manner.

Input: C {clustered code segment}, *Shuffle* {a method that takes a set as input and outputs a random ordering}
Output: a randomized code segment
  {Shuffle basic-blocks within the cluster.}
  $R \leftarrow \emptyset$
  for all $c \in C$ **do**
    $R \leftarrow R \cup \{Shuffle(c)\}$
  end for
  {Shuffle the order of clusters within the code segment.}
  return $Shuffle(R)$

indirect branch code, and lines 2 and 5 of the return code) to compare the target address against both bounds. A conditional jump immediately following each comparison transfers control to an abort function if any check fails. The abort function halts (and optionally re-randomizes and restarts) the program.

Some creativity is required to squeeze the lengthier legacy guards into a small number of chunks, while avoiding the introduction of gadgets (at chunk boundaries) abusable by attackers. For example, the short-jump instructions (at line 8 of the indirect branch code, and lines 3 and 6 of the return code) take circuitous routes to the abort function, but have shorter encodings than long-jumps, saving space. (Since the jumps are never taken during a policy-satisfying run, there is no performance downside.) The indirect branch code uses `ecx` as a temp register for indexing the BLT in lines 7 and 9, achieving shorter encodings of those instructions.

To prevent abuse of the second chunk boundary as a jump target during an attack, the legacy guard instructions are carefully arranged so that the second boundary splits an instruction's encoding, and the bytes falling after the boundary decode to an illegal opcode. In the return code listing, the *align_mask* argument (`0xF0`) decodes to an illegal opcode prefix. In the indirect branch code listing, the destination argument of the conditional jump in line 10 (abort) is chosen so that its first byte is an illegal instruction.[1] (As in MPX mode, the first chunk boundary requires no protection, since the instructions that follow it implement a fully guarded jump.)

C. Dynamic Randomization and Protection

The dynamic phase of our system proceeds at load-time, before the binary executes. In this phase the basic blocks are

[1]The argument to `jbe` is a 32-bit, little-endian, relative offset, so we prefix our abort function implementation with a 256-byte sled of NOPs, allowing jump offsets to it to have an arbitrary least-significant byte (e.g., `0xF4`).

Algorithm 4 $CreatePortal(N, t)$: Create a portal to t from nexus N.

Input: N {a nexus}, t {target address}
Output: Add a direct jump to t in the first available slot in N, and returns its address.
  for $i = 1$ **to** $Capacity(N)$ **do**
    if $N[i]$ is an empty slot **then**
      $N[i] \leftarrow CreateDirectJump(t)$
      return $Address(N[i])$
    end if
  end for
  return $null$ {Return null if insufficient capacity.}

Algorithm 5 $CreateAllPortals(C)$: Fill each nexus with portals to targets until (a) there are no more targets for that branch, or (b) the nexus capacity is reached.

Input: C {the ordered set of clusters, post randomization}
Output: Fill nexuses with portals.
  for all $c \in C$ **do**
    for $i = 1$ **to** $Capacity(c.nexus)$ **do**
      $b \leftarrow GetParentBranch(c)$
      $f \leftarrow FarthestTarget(c, b)$
      $q \leftarrow CreatePortal(c.nexus, f)$
      if $q \neq null$ **then**
        $UpdateTargets(b, f, q)$
      end if
    end for
  end for

randomized for diversity, bounds on indirect branches are fixed, and bounds ranges are further minimized. O-CFI uses a runtime library for this purpose, which it injects into the *Import Address Table (IAT)* of the rewritten binary during the static phase.

The Windows load order guarantees that all statically linked libraries initialize before the modules that link to them. The dynamic phase is thus carried out by the intermediate library in its initialization code. Algorithm 2 describes the order of steps taken by the initialization code.

First, the two-step process detailed in Algorithm 3 is used to randomize the code segment without affecting bounds ranges. Next, each nexus is populated using a greedy algorithm that creates portals to the farthest targets of its parent branch step-by-step, until its capacity is exhausted. Algorithm 5 shows the pseudo-code for this process while Algorithm 4 details how individual portals are created.

Once the targets of all branches have been finalized, direct branch operands are relocated to their correct locations and all jump-table entries in the `.told` section are updated with their new addresses. The BLT is also updated to reflect the new ranges for each branch, after which it is then moved to a random page of memory. Finally, the `gs` segment register is updated to point to the new base address of the BLT.

D. Platform Support and Infrastructure

We have implemented O-CFI for 32-bit versions of Windows (XP/Vista/7/8). Our O-CFI approach however, is not OS specific and is also applicable to 64-bit versions of Windows as well as Linux and OS X.

The implementation consists of three parts: (i) a static rewriter, (ii) an intermediate library, and (iii) an API hooking

TABLE V. SPACE AND REWRITING OVERHEADS

Binary Program	Size Increase		Rewriting Time (s)
	File (%)	Code (%)	
gzip	134	72	4.07
vpr	118	59	6.71
mcf	142	75	3.48
parser	141	76	5.96
gap	152	85	1.32
bzip2	133	72	3.50
twolf	120	53	7.84
art	140	74	3.62
equake	134	70	3.62
mean	+137%	+71%	5.85s

utility. The rewriter is implemented as a 2600 line IDA Python program. IDA Python programs are parsed and interpreted by the Hex-rays IDA Pro 6.1 disassembler. One of IDA Pro's many uses is as a malware reverse engineering and de-obfuscating tool, and it features many powerful code analyses that heuristically recover program structural information without assistance from source code or debug information. Our system leverages these analyses to automatically distinguish code from data, and to identify indirect branches and possible targets of such branches.

The intermediary library consists of approximately 500 lines of C and hand-written, in-lined assembly code that facilitates run-time fine-grained randomization and subsequent portal creation. It also provides support for callback functions and dynamic linking. An additional 150-line configuration file itemizes all trusted callback registration functions exported by Windows libraries used by the benchmarks.

The API hooking utility adds the intermediary library to the PE's list of imported modules. To avoid expanding the size of the PE header (which could shift the positions of the binary sections that follow it), our utility simply changes the library name `kernel32.dll` in the import section of the name of our intermediary library instead. The intermediary library exports all `kernel32` symbols as forwards to the real `kernel32`, except for security relevant functions, which it exports as local replacements.

V. EVALUATION

We tested O-CFI with binaries from the SPEC2000 benchmark suite listed in Table V. All results detailed below were obtained on an Asus G53SW machine with 4GB of RAM and the Intel Core i7 2630QM processor.

A. Rewriting and Space Overheads

Table V reports the percentage increase in file size and code size in the rewritten binaries, as well as the time taken by O-CFI to rewrite each binary.

Our prototype rewrites about 60KB of code per second on average. In MPX mode, the secured binaries increase in size by an average of 137%, while the code segment size increases by about 71%. The statistics for legacy mode are similar. (Although there is a small difference in size between the MPX and legacy guard codes, it does not significantly affect the overall space overheads.)

Although O-CFI's size increase may appear substantial, we believe that in many cases trading increased memory usage for substantially better security at high runtime performance is a worthwhile exchange. Our design is therefore calibrated to favor security and speed over space overheads in common cases.

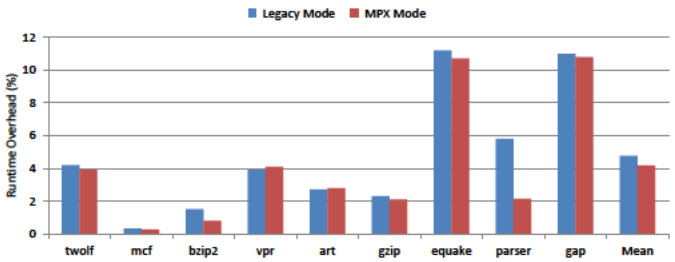

Fig. 4. O-CFI runtime overhead

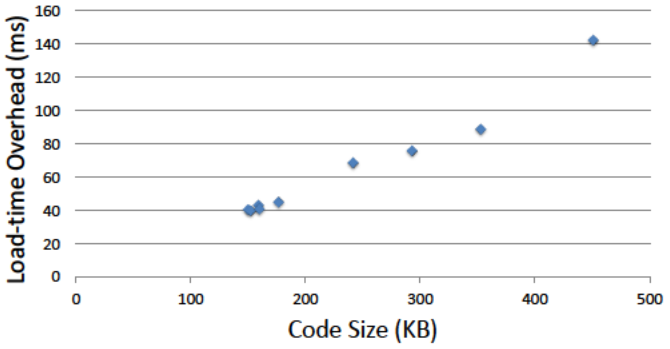

Fig. 5. O-CFI load-time overhead

B. Performance Overheads

Figure 4 shows the performance impact of O-CFI. We report overheads for both MPX and legacy mode guards for each binary. In both cases, they are measured as the mean percentage increase in runtimes between the rewritten and original binary over 20 trials. The average overhead across all benchmarks is 4.7% when using the legacy-mode guards shown rightmost in Table IV.

We cannot provide precise numbers on MPX-enabled hardware since such hardware is not yet available. However, since MPX instructions are treated as NOPs by the processor used in our experiments, and since long NOP instructions have roughly the same execution characteristics as ALU operations with register operands on Core i7 processors [14], we can use our current hardware to estimate the performance impact of MPX acceleration. We find that MPX instructions decrease the performance impact to 4.17%, and conclude that O-CFI is likely to benefit from MPX support from 2015 and onwards.

Both types of guards show similar characteristics across the benchmarks, exhibiting negligible overhead with `mcf` and `bzip2`, while `gap` and `equake` incur the largest overheads. A few benchmarks (`vpr` and `art`) exhibit marginally better performance with the legacy mode guards.

We also measured the overhead of the load-time phase, including cluster-based randomization, bounds range updating, and portal creation. These results are reported in Figure 5. Unsurprisingly, there is a linear relationship between the size of the code section and the overhead induced at load-time.

C. Security

To evaluate the security properties of O-CFI under our threat model, we evaluate the probability of chaining gadgets assuming full disclosure of the code segment. That is, we assume that the attacker knows the precise address of every gadget in the rewritten binary. Due to using segmented memory to access the BLT, an attacker is forced to guess the ranges

TABLE VI. GADGET CHAIN LENGTHS ACROSS SPEC BENCHMARKS

Gadget Chain Size	Chance (%)
2	2.0
3	0.8
4	0.01
5	–

of each gadget. (Section III-B examines the probability of guessing the address of the BLT itself.)

Under these assumptions, we carry out two experiments. First, we evaluate the statistical likelihood of chaining gadgets without violating branch bounds. Second, we attempt to use post-randomization gadget addresses to manually craft practical attacks against rewritten binaries, and evaluate how O-CFI resists implementation disclosure in practice.

1) Chaining Gadgets: We use the Mona ROP-generation tool [9] to identify all gadgets in a secured binary after the load-time phase, simulating full disclosure of the code segment. To simulate an attacker's attempts to guess branch ranges, payload chains are created by randomly selecting gadgets from the discovered gadget set. When a selected gadget falls outside the bounds of the previous gadget in the chain, the chain is terminated and its length is considered the payload length for that run. By repeating this process over multiple runs, we estimate the probability of successfully crafting payloads of various lengths.

This procedure estimates the probability of success of disclosure attacks resembling JIT-ROP, where a failure (such as a #BR violation) results in program termination and subsequent re-randomization of code and re-selection of bounds ranges.

BROP-like attacks, on the other hand, exploit the lack of re-randomization when binaries are respawned via the `fork` system call. A crucial advantage of our system against such attacks is the fact that a bounds violation does not automatically crash the program; rather, it invokes the registered #BR exception handler. The exception handler responds to the attack by forcing re-randomization, leaving BROP attacks on the same footing as JIT-ROP attacks.

In our experiments, we ran 100,000 runs of the experimental methodology (for each benchmark), and then measured the mean chance of success at crafting payloads of increasing lengths. This evaluates the security of O-CFI against disclosure attacks. The results are reported in Table VI.

Disclosure attacks in our experiments are unsuccessful at chaining more than four gadgets from any of the binaries. Moreover, the chance of chaining even a four-gadget payload is about 0.01% on average—a strong indicator that O-CFI offers powerful probabilistic protection against such attacks.

2) Crafting Practical Attacks: Mona is capable of building practical gadget chains based on a heuristic search. In particular, it can look for the gadgets necessary for attackers to (i) gain execute permissions, (ii) disable DEP, and (iii) allocate a new page with execute permissions.

To evaluate security against a more practical disclosure attack, we leveraged this capability, and tried to build chains using gadgets from both the original and rewritten binaries. For the rewritten binaries, we filtered any chains that resulted in bounds range exceptions.

Mona found partial chains (i.e., chains that incorporate some additional gadgets from dynamically linked modules) for all original binaries, and found full chains for some. However, no complete or partial chains could be found for any of the rewritten binaries. This provides additional evidence of the effectiveness of O-CFI against implementation disclosures.

TABLE VII. BOUNDS RANGE REDUCTION FACTORS WITH PORTALS

Binary Program	Nexus Capacity						
	3	6	9	12	15	18	∞
`gzip`	3.11	4.34	5.01	5.99	6.96	8.60	288.19
`twolf`	2.70	3.61	4.42	5.51	6.82	7.96	310.67
`vpr`	2.75	4.17	5.54	6.95	9.23	10.68	287.79
`gap`	2.11	2.57	3.05	3.49	3.94	4.37	255.46
`equake`	2.90	5.73	6.93	10.47	12.94	16.94	352.73
`art`	3.33	5.22	6.80	8.60	13.55	16.93	368.97
`mcf`	3.37	6.09	6.86	9.58	12.63	18.94	353.72
`bzip2`	2.87	4.38	6.32	7.43	8.52	13.34	277.88
`parser`	2.87	4.65	5.38	6.05	6.91	9.15	275.49
median	2.89	4.53	5.59	7.12	9.05	11.88	307.88

D. Portal Efficacy

We also tested the O-CFI's effectiveness at reducing bounds ranges. Table VII shows the factor by which average bounds sizes reduce as the nexus capacity varies from 3 to 18, and when the capacity is left unbounded. The bounds for each case are compared against a baseline binary that has a nexus capacity of zero. Across most binaries, with the exception of `gap`, the bounds sizes decrease roughly linearly as the number of portals increases. When the capacity is left unbounded, the bounds reduce by a factor of almost 308. In other words, the average bounds range for a binary with unbounded nexus capacities is only about 0.3% of the range for a binary that does not use portals.

As mentioned earlier, our implementation uses a nexus capacity of 12, which reduces bounds by a factor of about 7, while only marginally affecting code size or runtime performance. Figure 6 is a histogram of bounds sizes across all the benchmarks when the capacity is set to 12, with the counts measured on a logarithmic scale. Although there is a fairly wide variance in ranges, the overwhelming majority of bounds have span less than 15K in size.

E. Security against Theoretical Full-Knowledge Attack

In the previous section, we evaluated whether a ROP generation tool (Mona) can construct effective attacks for attackers who have not located the BLT (cf., §II-B). Although we protect the BLT from memory disclosure, we now consider an extraordinarily capable attacker who either (a) discovers all code pages, disassembles their contents, and infers all bounds from full knowledge of the program control flow, or (b) somehow locates and reads the BLT.

To the explore such an attacker's capabilities, we extended and adapted the Frankenstein mutation engine [31] to search for a ROP chain that implements the `VirtualAlloc` or `VirtualProtect` payloads from §V-C2 without violating any bounds. Frankenstein uses a constraint-solving algorithm to find gadget chains that realize a user-specified goal state. The goal processor state for our payloads was expressed as the stack layout needed for a protection-disabling system API call. Our tool has basic semantic understanding of a subset of x86 instruction sequences, mainly pertaining to their effect on the stack. It leverages this understanding to search for a satisfying sequence of gadgets from a given gadget pool.

When testing against a binary, the gadget pool is initialized to the set of gadgets found by Mona. Finally, for each runtime-

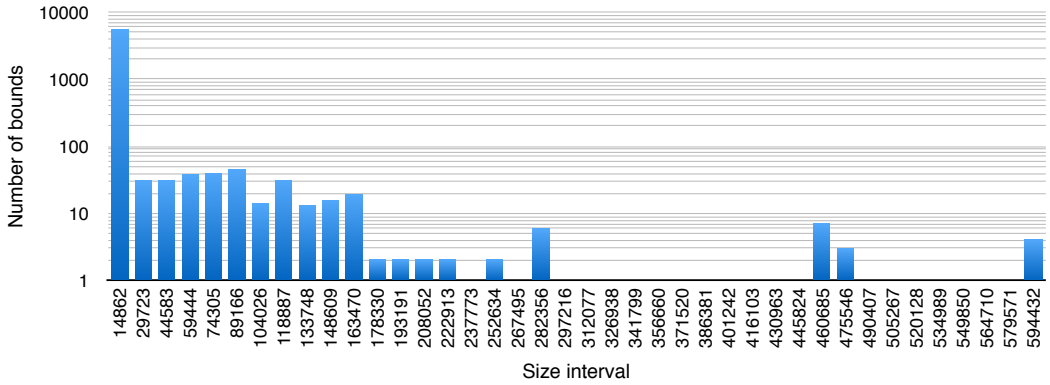

Fig. 6. Bounds range histogram for a nexus capacity of 12. The vast majority of bounds have span under 15K.

randomized layout that the tool is provided, the BLT-permitted range for each indirect branch is added as a constraint.

We used the tool on each of the SPEC2000 binaries, across 100 randomized code layouts each. Our results mirror the practical payload tests—although we found partial chains for the original binaries, we were unable to find any ROP chains that pass the bounds checks in binaries protected by O-CFI.

VI. Discussion

A. Branch Range Entropy

A bounds-guessing disclosure attack succeeds when every gadget falls within the range of the indirect branch in the preceding gadget. If a binary contains b indirect branches split across c clusters, and every indirect branch represents a potential gadget, each cluster contains on average b/c gadgets.

Given a randomly chosen initial gadget, the chances of choosing a second gadget that falls within the range of the first is $\frac{b/c}{b}$ or $1/c$. Thus, the chances of creating a payload with n gadgets becomes $(1/c)^{n-1}$. The average value of c across the SPEC binaries is 23. Thus, we find that the chance of creating a bounds-adherent payload falls below 0.01% when $n > 4$. This supports the experimentally obtained values for disclosure attacks in Table VI.

Rather than guessing branch ranges, an implementation disclosure attack could try to guess the location of the BLT in at attempt to recover all bounds, so that it may craft a payload in a more guided fashion. As discussed in §III-B, the probability of doing so falls as low as $1/2{,}147{,}483{,}648$ on x86-64 systems. Attackers who beat the odds and manage to locate the BLT still face the daunting challenge of leveraging the leaked information to craft a gadget sequence that (i) is not bounds-violating, and (ii) expresses a meaningful payload. Section V-E demonstrates that this is quite difficult given the sparsity of control-flows allowed by O-CFI.

B. Control-flow Obfuscation

Section V-E evaluates adversaries who use implementation disclosures, binary reverse-engineering, and control-flow fingerprinting to infer and recover the complete control-flow graphs of victim programs, and it concludes that such attackers still cannot reliably craft significant-length gadget chains. To further frustrate such efforts, O-CFI could be coupled with code obfuscation and anti-piracy strategies that hamper effective reverse-engineering. For example, instruction-level diversification (e.g., register allocation and instruction schedule randomization), code duplication, opaque predicates [28], control-flow flattening [44], and non-readable code pages [3] are all supportable by O-CFI.

C. External Module Support

O-CFI allows secured binaries to call library functions, and fully supports rewriting of libraries. We here discuss this capability in the context of Microsoft Windows libraries, but the approach generalizes to Linux as well.

1) Calling external modules from secured binaries: The Import Address Table (IAT) of secured binaries is set non-writable throughout program execution, preventing attackers from corrupting it to redirect control-flow to arbitrary locations. As such, calls made to external modules through the IAT need not be constrained by O-CFI, and work correctly unchanged.

Most binaries also access external modules using dynamic linking system calls, such as `LoadLibrary` and `GetProcAddress` on Windows. We use *trampolines* [48] to support such calls. In particular, the intermediary library that performs load-time randomization also intercepts all calls to the OS's dynamic linking API. The interception dynamically loads the requested library, but returns a chunk-aligned pointer to a trampoline within the caller's address space instead of the address of the requested function. When the trampoline is called, it safely transfers control to the library, ensuring compatibility between dynamically linked O-CFI and non-O-CFI binaries.

2) Securing Libraries: Libraries can be secured almost identically to program main modules. The only difference is that the returns of exported functions need special bounds that permit cross-module control-flows. O-CFI therefore creates a return-portal for each exported function and exports its address. This way, library returns become intra-module, and their bounds can be set to the range of all locally identified targets plus the address of the exported portal.

D. Approach Limitations and Future Work

Our prototype implementation of O-CFI relies on the static-rewriting phase to identify and protect all branch ranges. As such, it is unable to secure code that is generated dynamically. Consequently, although our prototype is compatible with binaries that generate JIT-code, it does not protect the JIT-compiled code. However, prior work has shown that both diversity and CFI can be successfully and efficiently applied at runtime to JIT-compiled code [21, 32]. We believe the same or similar strategies suffice to realize O-CFI for JIT code.

Our O-CFI prototype implementation is presently incompatible with Windows Component Object Model (COM). COM uses binary reflection to dynamically inform loading modules of the methods a COM library supports. Once the COM library is loaded, calls to its methods are implemented as indirect calls.

Because indirect calls in O-CFI binaries are masked to chunk-aligned targets for safety, and because any dynamically loaded COM module has function entry points that are not located at these boundaries, any attempt to jump to one of these functions crashes the binary. Supporting COM requires extending our implementation with a mechanism that dynamically creates (chunk-aligned) trampolines for the functions of loaded COM modules. This is reserved for future work.

A compiler-based implementation of O-CFI has access to a more precise control-flow graph of the target binary than is available to our COTS-compatible, binary-level prototype. Because completely accurate disassembly of x86 code is, in general, impossible [8, 23], our binary-level prototype is prone to false-negatives when identifying branch targets. (False-positive are harmless, at worst contributing to an increased bounds range for that branch.) Such false-negatives lead to runtime #BR violations for legitimate control-flows. This is a standard limitation of source-free CFI, and is addressable through improved disassembly heuristics (e.g., [47]).

VII. Related Work

Our work combines aspects of two previously separate areas: control-flow integrity and software diversity. We discuss each of these separately.

A. Software Diversity

Cohen was first to describe software diversity as a defense mechanism [8]. Forrest subsequently demonstrated stack-layout randomization as a defense against stack smashing [15]. Subsequent work on artificial diversity is extensive; Larsen et al. [27] provide an overview.

The work most closely related to ours is the concurrently developed Oxymoron system [2]. Oxymoron uses a pure code-randomization approach to resist JIT-ROP attacks. By generating code that does not contain any direct references to other code pages, it impedes the JIT-ROP attack step that recursively infers new code page addresses by disassembling leaked code pages. Instead of direct code references, inter-page references use an indirection table similar to our BLT, whose base address is stored in an x86 segment register to protect it from accidental disclosure.

However, Oxymoron allows raw code addresses to flow into registers, into the heap, and onto the stack. As a result, it is vulnerable to buffer over-read attacks that disclose the stack and heap contents [42], which can hold a wealth of raw return addresses and function pointers, each of which potentially reveals a 4K page of gadgets for attackers to abuse. Concurrent work has exploited this weakness to bypass Oxymoron by harvesting code pointers from vtables in victim C++ programs [10]. The XnR approach, which prevents reads to code pages, shares this weakness [3]. In contrast, we pessimistically assume that code addresses will eventually leak to attackers (because they are stored in too many places). Instead, O-CFI conceals the control-flow policy graph, whose details can be more easily protected from disclosure.

Giuffrida et al. [17] present a comprehensive compiler-based software diversification approach that allows live re-randomization. The resulting overheads depend on the re-randomization frequency. Snow et al. [40] report that JIT-ROP attacks can run in as little as 2.3 seconds. Re-randomizing every two seconds, however, adds an overhead of about 20%.

The foundations for the basic block randomization portion of our implementation are based upon the STIR system [45]. STIR analyzes and rewrites binaries ahead of time to make them self-randomizing (or self-stirring). At load-time, a small randomization stub permutes the basic block and function layout inside the binary. To account for disassembly errors, the original code is loaded as non-executable data, allowing the stirred binaries to compensate for disassembly errors.

B. Control-Flow Integrity

In its most precise form, Control-Flow Integrity CFI [1] confines indirect branches to flow only to a precise set of statically identified targets for each branch. CFI has yet to see widespread adoption in the industry. We believe the two main reasons for this to be (i) difficulty reconstructing an accurate CFG for a binary without access to source code or debug symbols, neither of which are readily available for the large majority of COTS binaries, and (ii) higher overheads than solutions based on code randomization.

A number of low-overhead solutions have been proposed which impose less strict integrity checks on program executions. These include ROPecker [7], ROPGuard [16] and kBouncer [33]. ROPecker and kBouncer use the x86 last branch record (LBR) register set to accomplish their checks. kBouncer, for instance, performs CFI validation on the LBR during any Windows API invocation, and ensures that all returns all are call-preceded (i.e., that each return address points to an instruction that follows a call instruction). ROPecker creates an offline gadget database, which is then compared at runtime with LBR entries to detect attacks. ROPGuard also performs CFI validation on Windows API calls. Like kBouncer, it requires that return addresses are call-preceded. It also verifies that the memory word before each return address is the start address of the API function.

CFI for COTS binaries [50] is another proposed coarse-grained CFI solution. This technique can be applied to binaries without access to source code or debug information. It relies on a static disassembly step where all potential branch (and return) targets are identified, and all indirect branches are instrumented with code that jumps to a CFI validation routine. The validation routine ensures that target and return addresses are either call-preceded, or belong to the set of statically identified targets.

Similarly, Compact Control Flow Integrity and Randomization (CCFIR) [49] applies coarse-grained CFI to binaries without source code or debug information (but with relocation information). In this technique, permissible targets for indirect branches are collected into a separate Springboard section, and indirect branches are only allowed to flow into the Springboard. CCFIR also incorporates some elements of code-randomization—target entries are placed at random locations within the Springboard. Although this confers an additional degree of security against traditional ROP attacks, disclosure attacks are able to read the full contents of the Springboard, nullifying its advantages against that class of attacks.

Davi et al. [11] test these coarse-grained CFI solutions and show that they fail to adequately secure binaries against ROP attacks.

Forward CFI [43] and SafeDispatch [25] are two recent compiler-based CFI solutions. Forward CFI protects binaries by inserting validation checks for all forward-edge control flows. This solution is only intended to secure forward control-flows, and does not protect against attacks that rely purely on return-terminated gadgets.

SafeDispatch protects C++ binaries from *virtual table hijacking* by recompiling binaries with a modified C++ compiler that instruments all virtual method call sites with runtime checks that ensure that all method calls jump to valid implementations during execution. Additionally, SafeDispatch only protects virtual method calls, leaving binaries vulnerable to ROP attacks that rely on modified return addresses on the stack.

VIII. Conclusions

CFI and artificial software diversity are well-established, complementary strategies for protecting software against code-reuse attacks, including ROP attacks. Recent advances in offensive security have alarmingly demonstrated how to bypass both: The security relaxations introduced by coarse-grained CFI to achieve acceptable performance are exploitable by skillful control-flow hijacking, and implementation disclosure vulnerabilities can be leveraged to derandomize even fine-grained artificial diversity defenses.

O-CFI combines and extends both CFI and fine-grained diversity to address this dual threat of code-reuse and implementation disclosure attacks. To do so, we reformulate CFI as a bounds-checking problem, and repurpose fine-grained binary code randomization to diversify and conceal the exploitable edges of the protected program's control-flow graph. As a result, O-CFI can protect software even against attackers who have complete read-access to the randomized program code.

Our prototype implementation demonstrates that O-CFI can be effectively applied to protect legacy binaries without source code, and experimental evaluation exhibits performance overheads of just 4.7% on legacy processors. Performance is expected to be even higher (c. 4.17% overhead) on future-generation processors, since our bounds-checking implementation centers around Intel MPX instructions that will be hardware-accelerated on forthcoming Intel-based processors.

Acknowledgments

We thank Julian Lettner for benchmarking assistance, and Lucas Davi, Andrei Homescu, Christopher Liebchen, Matt Miller, and the anonymous reviewers for their insightful comments and suggestions.

This material is based upon work partially supported by National Science Foundation (NSF) CAREER award #1054629, Office of Naval Research (ONR) award N00014-14-1-0030, Air Force Office of Scientific Research (AFOSR) award FA9550-14-1-0173, an Industry-University Collaborative Research Center grant from Raytheon Company, Defense Advanced Research Projects Agency (DARPA) contracts D11PC20024 and N660001-1-2-4014, and by gifts from Mozilla Corporation and Oracle Corporation. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the above supporters, their contracting agents, or any other agency of the U.S. Government.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Information and System Security (TISSEC)*, vol. 13, no. 1, 2009.

[2] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proc. 23rd Usenix Security Sym.*, 2014, pp. 433–447.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*, 2014, pp. 1342–1353.

[4] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proc. 35th IEEE Sym. Security & Privacy (S&P)*, 2014, pp. 227–242.

[5] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Sym. Information, Computer and Communications Security (ASIACCS)*, 2011, pp. 30–40.

[6] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23rd Usenix Security Sym.*, 2014, pp. 385–399.

[7] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. 21st Annual Network & Distributed System Security Sym. (NDSS)*, 2014.

[8] F. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, no. 6, pp. 565–584, 1993.

[9] Corelan Team, "Mona," 2014, https://github.com/corelan/mona.

[10] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proc. 22nd Network and Distributed Systems Security Sym. (NDSS)*, 2015.

[11] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. 23rd Usenix Security Sym.*, 2014, pp. 401–416.

[12] J. DeMott, "Bypassing EMET 4.1," Bromium Labs, 2014, http://labs.bromium.com/2014/02/24/bypassing-emet-4-1.

[13] C. Evans, "Exploiting 64-bit Linux like a boss," 2013, http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html.

[14] A. Fog, "Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," 2014, http://www.agner.org/optimize/instruction_tables.pdf.

[15] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proc. Workshop Hot Topics in Operating Systems*, 1997, pp. 67–72.

[16] I. Fratrić, "Runtime prevention of return-oriented programming attacks," University of Zagreb, 2012, http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf.

[17] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Security Sym.*, 2012, pp. 475–490.

[18] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. 35th IEEE Sym. Security & Privacy (S&P)*, 2014, pp. 575–589.

[19] 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 *Proc. 23rd Usenix Security Sym.*, 2014, pp. 417–432.

[20] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in Turing-complete return-oriented programming," in *Proc. 6th USENIX Workshop Offensive Technologies (WOOT)*, 2012, pp. 64–76.

[21] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: Transparent code randomization for just-in-time compilers," in *Proc. 20th ACM Conf. Computer and Communications Security (CCS)*, 2013, pp. 993–1004.

[22] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proc. 11th IEEE/ACM Int. Sym. Code Generation and Optimization (CGO)*, 2013, pp. 1–11.

[23] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *The Computer J.*, vol. 23, no. 3, pp. 223–229, 1980.

[24] Intel, "Introduction to intel memory protection extensions," https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

[25] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ virtual calls from memory corruption attacks," in *Proc. 21st Annual Network & Distributed System Security Sym. (NDSS)*, 2014.

[26] N. Joly, "Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013)," VUPEN Vulnerability Research Team (VRT), 2013, http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php.

[27] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. 35th IEEE Sym. Security & Privacy (S&P)*, 2014, pp. 276–291.

[28] A. Majumdar and C. Thomborson, "Securing mobile agents control flow using opaque predicates," in *Proc. 9th Int. Conf. Knowledge-based Intelligent Information and Engineering Systems*, vol. 3, 2005, pp. 1065–1071.

[29] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proc. 15th USENIX Security Sym.*, 2006.

[30] Microsoft, "Enhanced mitigation experience toolkit," https://www.microsoft.com/emet, 2014.

[31] V. Mohan and K. W. Hamlen, "Frankenstein: Stitching malware from benign binaries," in *Proc. 6th USENIX Workshop Offensive Technologies (WOOT)*, 2012, pp. 77–84.

[32] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*, 2014, pp. 1317–1328.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Security Sym.*, 2013, pp. 447–462.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. 20th USENIX Security Sym.*, 2011.

[35] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*, 2014, pp. 54–65.

[36] F. J. Serna, "The info leak era on software exploitation," Black Hat USA, 2012.

[37] ——, "CVE-2012-0769, the case of the perfect info leak," Google Security Team, 2012, http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[38] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, 2004, pp. 298–307.

[40] K. Z. Snow, F. Monrose, L. V. 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 *Proc. 34th IEEE Sym. Security & Privacy (S&P)*, 2013, pp. 574–588.

[41] A. Sotirov, "Heap feng shui in JavaScript," Black Hat Europe, 2007, https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[42] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proc. 2nd European Workshop System Security (EUROSEC)*, 2009, pp. 1–8.

[43] 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 *Proc. 23rd USENIX Security Sym.*, 2014.

[44] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," University of Virginia Charlottesville, Tech. Rep., 2000.

[45] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. 19th ACM Conf. Computer and Communications Security (CCS)*, 2012, pp. 157–168.

[46] ——, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, 2012, pp. 299–308.

[47] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu, "Shingled graph disassembly: Finding the undecidable path," in *Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, 2014, pp. 273–285.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Sym. Security & Privacy (S&P)*, 2009, pp. 79–93.

[49] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. 34th IEEE Sym. Security & Privacy (S&P)*, 2013, pp. 559–573.

[50] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22nd USENIX Security Sym.*, 2013, pp. 337–352.