Fine-Grained Control-Flow Integrity for Kernel Software

Xinyang Ge*, Nirupama Talele*, Mathias Payer†, and Trent Jaeger*
**The Pennsylvania State University*
†*Purdue University*

***Abstract*—Modern systems assume that privileged software always behaves as expected, however, such assumptions may not hold given the prevalence of kernel vulnerabilities. One idea is to employ defenses to restrict how adversaries may exploit such vulnerabilities, such as Control-Flow Integrity (CFI), which restricts execution to a Control-Flow Graph (CFG). However, proposed applications of CFI enforcement to kernel software are too coarse-grained to restrict the adversary effectively and either fail to enforce CFI comprehensively or are very expensive.**

We present a mostly-automated approach for retrofitting kernel software that leverages features of such software to enable comprehensive, efficient, fine-grained CFI enforcement. We achieve this goal by leveraging two insights. We first leverage the conservative function pointer usage patterns found in the kernel source code to develop a method to compute fine-grained CFGs for kernel software. Second, we identify two opportunities for removing CFI instrumentation relative to prior optimization techniques: reusing existing kernel instrumentation and creating direct transfers, where possible. Using these insights, we show how to choose optimized defenses for kernels to handle system events, enabling comprehensive and efficient CFI enforcement.

We evaluate the effectiveness of the proposed fine-grained CFI instrumentation by applying the retrofitting approach comprehensively to FreeBSD, the MINIX microkernel system, and MINIX's user-space servers, and applying this approach partly to the BitVisor hypervisor. We show that our approach eliminates over 70% of the indirect targets relative to the best current, fine-grained CFI techniques, while our optimizations reduce the instrumentation necessary to enforce coarse-grained CFI. The performance improvement due to our optimizations ranges from 51%/25% for MINIX to 12%/17% for FreeBSD for the average/maximum microbenchmark overhead. The evaluation shows that fine-grained CFI instrumentation can be computed for kernel software in practice and can be enforced more efficiently than coarse-grained CFI instrumentation.

1. Introduction

A fundamental trust assumption is the integrity of a system's trusted computing base. In modern systems, hypervisors, microkernels, and operating system kernels are part of a system's trusted computing base, but they all service requests from untrusted parties. As a result, protecting the integrity of such kernel software[1] in the face of such threats is fundamental to system security.

Unfortunately, kernel vulnerabilities are still a common occurrence, and several recent vulnerabilities have enabled adversaries to control kernel execution. As of August 2015, 1301 Linux kernel CVEs have been reported [5]. Some of these vulnerabilities directly permit arbitrary code execution in the Linux kernel via buffer overflows (e.g., CVE-2014-2523), errors in packet handling (e.g., CVE-2015-1465) and errors in privilege level transitions (e.g., BadIRET). In addition, many vulnerabilities were reported for errors in pointer handling that could have led to arbitrary code execution. In spite of the deployment of W$\oplus$X protection which thwarts direct code-injection attacks by preventing execution on data memory (e.g., data execution prevention or DEP [8]), such kernel vulnerabilities may allow adversaries to launch control-flow hijacking attacks against the kernel, including *return-oriented attacks* [35] to create rootkits [24], enabling adversaries to perform arbitrary malicious computations.

To prevent adversaries from launching control-flow hijacking attacks, researchers have proposed enforcing *control-flow integrity* [6]. Control-flow integrity (CFI) aims to restrict the execution of a program to its control-flow graph (CFG). However, there are several challenges in enforcing CFI in practice. First, the prevalence of *indirect control transfers* in programs, such as indirect calls (i.e., function pointers) and returns, makes it difficult to compute fine-grained CFGs. In addition, the initial performance overheads reported for CFI implementations were significant. As a result, researchers have explored different levels of precision in the CFG enforced, including *lightweight CFI approaches* [15, 20, 32] that only authorize operations at critical points (e.g., system calls) and *coarse-grained CFI approaches* [17, 44–46] that only enforce call and return targets (CCFIR enforces two sets of return targets). However, several recent exploits [12, 19, 23] have shown that current lightweight and coarse-grained CFI implementations are insufficient to block control-flow hijacking attacks.

Researchers have also proposed stricter CFI policies, but these methods have significant limitations when applied to kernel software. Modular CFI [30] and forward-edge

[1] We will refer to conventional kernels, hypervisors, microkernels and their user-space servers collectively as "kernel software" in this paper.

CFI [41] both predict call targets by matching function pointers with function signatures to reduce the size of target sets. However, such signatures are not always available given the presence of variable-argument functions and assembly functions in the kernel code. In addition, signature-based approaches may still result in both false negatives, when a non-target function happens to have the same signature, and false positives, when a function address is assigned to a function pointer of a different signature. Both of these cases are common in kernel code.

Finally, kernel software presents additional challenges in CFI enforcement. In particular, kernel software must process asynchronous system events, such as system calls, interrupts, and exceptions. HyperSafe [44] was the first project to explore CFI enforcement in kernel software, focusing on hypervisors. However, HyperSafe did not prevent attacks on kernel exits to user-space (i.e., ret2usr [25]) and does not prevent the kernel from inadvertently changing its event handling configuration. KCoFI [17] alternatively provides a complete implementation for managing event handling based on the Secure Virtual Architecture [18], but the resultant cost of CFI enforcement is quite high (over 100%).

In this paper, we aim to show that fine-grained CFI enforcement for kernel software is possible, can be more efficient than coarse-grained enforcement, and can be applied comprehensively to kernel software, raising the bar for adversaries that wish to launch control-flow hijacking attacks. To achieve this goal, we first compute a fine-grained CFG for kernel code. Our mechanism supports both C and assembly regions (which are common in kernel code). The insight for producing a fine-grained CFG for kernels is that we find that kernel code uses function pointers in a limited way, rarely creating data pointers to memory locations (variables, array elements, or fields) assigned function pointers. This enables us to design a static analysis that covers large kernel code bases with few exceptions requiring manual intervention. To enforce the computed CFG, we choose restricted pointer indexing, which was originally proposed by HyperSafe, as the default enforcement instrumentation, but reduce overhead by specializing the instrumentation based on the number of legal targets, and reuse checks that are already available in the kernel code. Finally, we also develop a design that enables comprehensive and efficient CFI enforcement in kernel software by reasoning about how system events are processed. As a result, we have eliminated over 70% of targets on both FreeBSD and MINIX relative to the current fine-grained CFI, while our implementation incurs 1.82% performance overhead on FreeBSD and 0.76% overhead on MINIX on macrobenchmarks, and 11.91%/42.03% (average/maximum) and 2.02%/5.64% overhead on microbenchmarks.

Contributions. In this paper, we develop a mostly-automated approach to produce and enforce fine-grained CFI policies comprehensively for kernel software like the FreeBSD kernel, the MINIX microkernel and its user-space servers for Intel x86 platforms. In particular, we make the following contributions:

- We develop an automated method that leverages simple kernel code patterns in their use of function pointers (for indirect calls) and function structures (for returns) to produce a fine-grained CFG that includes all targets (or reports a failure to adhere to expected patterns). We apply our method to $\sim$10M SLoC of kernel software, including FreeBSD, MINIX, its user-space servers, and BitVisor, demonstrating its applicability. The resulting CFGs are restrictive, where over 90% of the indirect control transfers have ten or fewer authorized targets.
- We develop an automated method that uses the type of indirect control transfer and the target set size of each indirect control transfer to choose the most efficient instrumentation code. In addition, we define a set of invariants applicable to enforcing CFI comprehensively despite system event handling in kernel software and describe how those invariants may be achieved efficiently.
- We evaluate our fine-grained CFI enforcement by applying the proposed techniques to the FreeBSD kernel, the MINIX microkernel and its user-space servers. We find that our instrumentation is both effective, eliminating over 99% targets that are allowed by a typical coarse-grained CFI implementation and over 70% of the call targets that are allowed by signature-based methods with no false positives detected, and efficient, incurring less overhead than a comparable coarse-grained CFI implementation.

2. Background

In this section, we outline the general idea of code-reuse attacks and how kernels pose additional risks in being attacked. Next, we look at the concept of control-flow integrity and reason about why and how current implementations over-approximate the control-flow graph to which program execution is restricted.

2.1. Code-Reuse Attacks

Return-oriented programming (ROP) [35] is a class of code-reuse attacks where adversaries combine short code sequences ending in `ret` instructions found within the victims' binary programs to perform malicious actions. Researchers have proven that this attack vector is Turing-complete, making it a generally applicable threat to systems protected by code-injection defenses such as W$\oplus$X. Similar code-reuse attacks include jump-oriented programming [9], which chain together code sequences that end in an indirect branch, and call-oriented programming [12], which chain together code sequences that end in an indirect call.

Generally, code-reuse attacks need to divert the original control flow to instructions of the adversary's choice, which requires the adversary to control the targets of control transfer instructions throughout an attack. Programs usually contain two types of control transfers: *direct* and *indirect*. Direct control transfers have fixed targets that are embedded into the program's code, which cannot be modified when W⊕X defenses are deployed, so they cannot be controlled to direct attacks. Instead, all these code-reuse attacks hijack the program's control flow by leveraging indirect control transfers, whose targets are determined at runtime.

Kernel software is often designed purposely using indirect control transfers to improve flexibility. For instance, kernels store all the addresses of interrupt handlers in a hardware-defined data structure. From this perspective, an interrupt acting as an entry to the kernel is a special indirect control transfer. Correspondingly, exits from the kernel are also indirect control transfers, of which interrupt and system call returns are two typical cases. In addition, kernel software often permits different configurations to use different code (e.g., for different architectures or feature specialization), defining function pointers that can be bound to the specialized code. If an adversary somehow controls any of these indirect control transfers, she can divert the kernel's control flow and potentially launch code-reuse attacks.

2.2. Control-Flow Integrity

Abadi *et al.* [6] proposed control-flow integrity (CFI), which enforces that a program's control transfers at runtime must adhere to the program's static control-flow graph (CFG). In a CFG, every node represents a basic block while an edge connecting two basic blocks indicates a legal control transfer from one to the other. However, due to the wide use of indirect control transfers, it is non-trivial to compute fine-grained program CFGs. In practice, CFI methods must over-approximate the number of permissible indirect control targets because missing any valid CFG edges may cause a legitimate control transfer to be rejected and a program execution to be incorrectly terminated. As a result, the proposed CFI methods often include extra CFG edges as shown by the dotted lines in Figure 1, particularly so-called *coarse-grained CFI approaches* [6, 17, 44, 46] that often just restrict transfer to either legal call targets (for calls) or legal return targets (for returns), which also leads to extraneous edges for the corresponding return instructions. The consequence of overapproximation is that this provides adversaries with more unintended control flows that may be utilized for attacks.

In order to compute a fine-grained CFG to reduce the set of allowed targets, two recent proposals [30, 41] use function signatures to identify function pointer targets. This *signature-based CFI approach* is simple, easy to implement, and can reduce the target set significantly. However, this method is not applicable to kernel software because it cannot match functions written in assembly, which are very common in kernel software (e.g., for architecture-specific code). In addition, this method can also cause false positives by excluding target functions whose type has been cast. We have seen instances of such behavior in kernel code.

To enforce CFI, researchers have proposed various approaches. One major concern in enforcing CFI is high performance cost, resulting in *lightweight CFI approaches* [15, 20, 32] that only authorize operations at critical points, such as on system calls and/or sensitive library calls. These methods leverage the *Last Branch Record* (LBR) [4] feature available on modern Intel processors to inspect the historical control flows, and kill the process if an active code-reuse attack is detected.

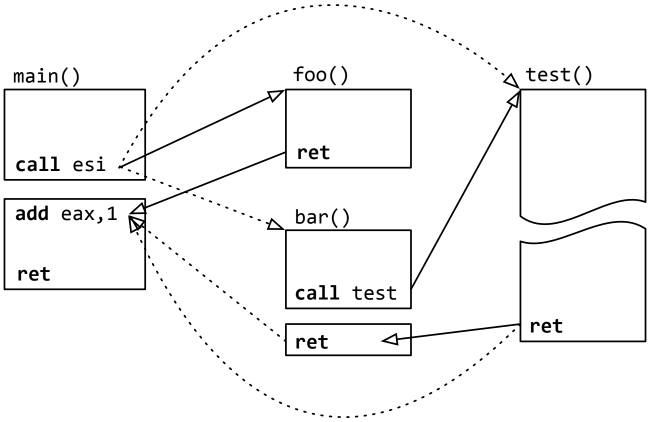

Figure 1: An over-approximated CFG

Alternatively, researchers propose to instrument the programs before each indirect control transfer [6, 17, 30, 33, 44–46]. In the original CFI proposal, Abadi *et al.* proposed to place a label before each allowed target and check if the label matches upon control transfers. However, in practice, researchers enforce *coarse-grained CFI*, where any legal call target may be used in any indirect call and any legal return target may be used for any return. In the two approaches where CFI has been applied to kernel software, HyperSafe and KCoFI, enforce coarse-grained CFI policies, although HyperSafe is capable of representing fine-grained CFI policies and even provided a solution to the *destination equivalence problem* in the original proposal.

However, researchers have shown that lightweight CFI and coarse-grained CFI approaches can be circumvented. The lightweight approaches are proven to be vulnerable to history flushing attacks [12]. The key idea behind this attack is to leverage the limited capacity of branch recording to clean up the history before the introspection routine is invoked. In addition, researchers found that adversaries can evade coarse-grained CFI approach by choosing allowed but unintended control transfers to launch attacks [11, 12, 19, 23]. In some cases, even fine-grained CFI can be circumvented by bending the control-flow along valid edges in the CFG [11]. Since signature-based CFI approaches are not applicable to kernel software for the reasons discussed above (and still quite coarse-grained), we need a new approach to guide the deployment of CFI enforcement for kernel software.

Once a fine-grained CFG can be computed, it must still be enforced comprehensively by the kernel software. The main additional problem is that kernel software must process various system events, system calls, interrupts, and exceptions, creating a new set of entry and exit points that must be protected and these events are asynchronous to the kernel. HyperSafe [44] recognized the need to protect the tables holding references to these event targets, but did not describe how to protect the all kernel values that locate these tables nor protect the exit. On the other hand, KCoFI [17] developed a comprehensive solution, but the cost of their solution is prohibitively high.

3. Security Model

We base our work on the following security model. We trust that the W⊕X protection is deployed in kernels by setting the code section as read-only and the data sections as non-executable. We also assume the kernel is benign but may contain vulnerabilities (e.g., memory-safety bugs) that enable an adversary to overwrite control data. However, we make the assumption that the kernel is free of attacks prior to the execution of the first user-space process. That is, we leave the detection of attacks at kernel boot, such as corruption of kernel images or loading of malicious code, to secure boot techniques [40] or authenticated boot techniques [21, 39]. Finally, we assume adversaries have no physical access to the machine, excluding hardware attacks from this work.

Thus, we assume that all threats originate from inputs that the kernel receives after booting, such as malicious inputs from user-space processes and malicious network packets. We focus on attacks that alter the control flow of the kernel software illegally at some indirect control transfer. Thus, data-only attacks are out of the scope of this work, including those data-only attacks that change the values of variables used in conditionals to redirect control flow within a procedure [7, 11, 13].

We assume the MMU configurations, such as mappings and permissions configured by system page tables, can be protected by system defenses such as Readactor [16], NICKLE [34] and SPROBES [22]. This is because, although the CFI-protected kernel restricts its execution to only authorized control flows even when compromised, the adversary may alter the system page table settings through *data-only* attacks. This could enable the adversary to modify existing code after disabling W⊕X protection, and hence allow her to perform arbitrary computation without the need to hijack control flows or reuse existing code. Previous work has shown that protecting MMU configurations can be done efficiently. NICKLE, a VMM-based system, reports ∼5% performance overheads when used to protect the kernel code integrity. SPROBES, a TrustZone-based mechanism that is used to protect Linux kernel code integrity on a different ISA (i.e., ARMv6) reports similar results (∼8%). In addition, Readactor [16] repurposes the use of virtualization hardware, i.e., extended page tables, available on modern Intel processors to enforce additional page permissions whilst incurring negligible performance overheads (2.5%). Given our system requires the kernel software to be statically linked (Section 4) to allow that all kernel code pages are identified offline, we believe their approach applies to our situation as well. However, we argue that, whatever MMU protection is used, its implementation and performance impact is orthogonal to the techniques discussed in this paper.

4. Solution Overview

Our goal in this work is to retrofit kernel software to enforce fine-grained CFI efficiently and comprehensively. We focus on kernel software because its integrity is fundamental to the integrity of the system at large and it is constrained in ways that enable computation of fine-grained CFGs.

In the first phase of our proposed solution, described in Section 5, we develop a method to produce accurate, fine-grained CFGs for kernel software. We hypothesize that we can collect an accurate set of indirect call targets for kernel code because we find that kernel code handles function pointers in restricted ways. In Section 5.1, we identify two simple constraints on the use of function pointer variables that we find that kernel code broadly obeys. For example, these constraints still allow kernel developers to use function pointers stored in arrays and structures, which is common. With these constraints, we develop a static taint analysis to identify the indirect call targets in kernel code. In Section 5.2, we describe how to compute return targets for kernel code. While computing return targets for source code is straightforward, assembly code does present some challenges for detecting legal return targets (e.g., due to tail-call optimization and fall-through functions), so we design a CFG analysis that can detect return targets accurately and comprehensively.

We find kernel software amenable to the application of CFI enforcement because it is often statically linked. Secure software deployments often demand static-linking to prevent adversaries from replacing critical software components at runtime. For example, the Linux Security Modules were originally loadable kernel modules, but such "modules" must now be statically linked into the kernel. This is because, kernel modules violate the W⊕X assumption in such a way that the kernel must first load them into writable memory pages and then mark them as executable. By corrupting the loaded modules in the vulnerable time window, an adversary can inject her own code and hence perform arbitrary computation without needing to reuse existing code. As a result, the approved code must be determined at load-time, enabling hardware-based methods [22, 34] to protect kernel code integrity by authorizing modifications of memory protections set by the kernel at boot time. As described in Section 3, we assume the presence of such defenses in this work, as opposed to increasing the complexity of our own methods to additionally enforce code integrity protection with CFI [17].

In the second phase of our proposed solution, we modify the kernel code to enforce the computed CFG. We find that two kinds of modifications are necessary. First, assuming the protection of kernel code integrity as discussed above, kernels still require further modifications to handle event processing (e.g., system calls and interrupts) in a manner compliant with the CFI enforcement at runtime. These issues have been investigated in the past in the HyperSafe [44] and

KCoFI [17] projects. However, HyperSafe does not address attacks on kernel exit nor how kernel code must be modified to ensure comprehensive enforcement. KCoFI, on the other hand, describes comprehensive method for controlling event processing, but at significant expense. In Section 6, we specify invariants that must be enforced to ensure that the kernel code is always invoked from legitimate entries and returns safely. Our aim is for comprehensive CFI enforcement given system event handling at low cost, which we achieve by removing all means for the kernel to modify event handling configurations, lightweight checking of exception handling, and non-preemptive kernel configurations[2].

In addition, given a fine-grained CFG computed according to the methods above, we develop a method to instrument the software to enforce the CFG at runtime as discussed in Section 7. While we use a standard form of instrumentation, called *restricted pointer indexing* [44] (see Section 7.1), as the default, we identify two opportunities for optimization. We find that kernel software often uses function pointers to express flexibility, so in many cases kernel software only uses one target for indirect control transfers. As a result, once we know that an indirect control transfer has only one target, we convert it to a direct control transfer, which requires no additional instrumentation. In addition, based on the type of addressing mode used, we find that we can reuse code already produced by the compiler or by manual assembly, again enabling the removal of unnecessary instrumentation.

5. Computing Control-Flow Graphs

In the first phase, we develop methods that compute a fine-grained CFG from kernel source code. We propose an algorithm for computing control transfer targets for indirect calls in Section 5.1, and discuss and solve the challenges in mapping calls to returns to compute the return targets in Section 5.2. We note that the use of indirect jumps in source code is limited to *switch* statements, whose targets are stored in jump tables and can be found trivially.

5.1. Computing Indirect Call Targets

We compute indirect call targets for kernel source code under constraints on the use of function pointer variables in the program code. Given these constraints, we design a static taint analysis to collect the functions that can reach each indirect call target that accounts for storing function pointers in complex data structures, such as arrays, structures, and unions. Based on our experience examining kernel source code, these constraints are followed broadly, enabling automated detection of targets. We evaluate the applicability of this method to kernel source code in Section 9.

In kernel source code, operations on function pointers are often limited to assignment and dereference. Thus, we propose two assumptions about operations on function pointers:

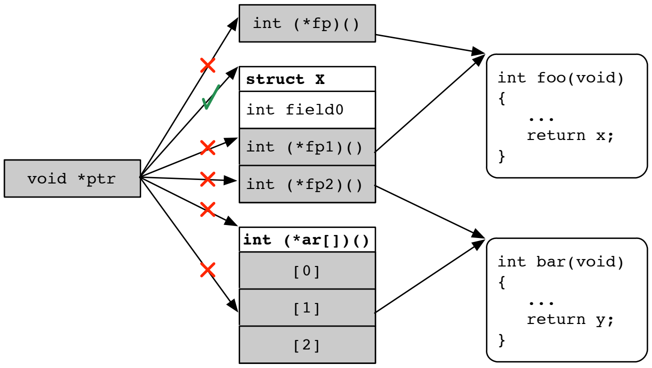

Figure 2: The assumption (A2) of the absence of data pointers to function pointers in kernel software

- (**A1**) The only allowed operation on a function pointer is assignment[3].
- (**A2**) There exists no data pointer to a function pointer.

There are a few important implications resulting from these assumptions. A1 limits the operations on function pointers, preventing them from being modified once assigned. In particular, this assumption precludes pointer arithmetic on function pointers. We believe arbitrary computations on function pointers are unlikely due to considerations such as readability, maintainability, and portability. A2 assumes the absence of data pointers to function pointers as illustrated in Figure 2. Note that A2 does not prohibit the presence of pointers to a structure that has function pointer fields. In fact, this is common in practice. For an array of function pointers, A2 only allows the array elements to be accessed by index from the array variable. Creating a pointer to the array or any of its elements will violate A2. We describe how we detect violations of these assumptions at the end of this section.

We use the following terminology to describe our algorithm. We define a *memory object* as a contiguous range of memory, such as a global/local variable, a function argument, or a dynamically allocated array. We refer a *function pointer* as a memory object whose content is an address of a function. We say a function f is a *valid target* for a function pointer p if and only if the content of the memory object indicated by p may be the address of f at runtime.

Our approach takes a function *f* as input and returns a set of function pointers for which *f* is a valid target as the output. To begin with, the approach taints all function pointers that are initialized with *f*. Note that a function pointer can be initialized either dynamically (e.g., assignment) or statically (e.g., global variables). Then we keep tainting function pointers to which a tainted function pointer is assigned. Because of assumption A1, we will not miss any function

[2] See Section 10 for discussion of the practicality of non-preemptive kernels and some trade-offs between preemptive and non-preemptive kernels.

[3] Other than dereferencing for calls, of course.

pointers tainted by function *f* by only tracking propagation via assignments.

Function pointers may either be referenced as a *variable*, an *array element*, or a *structure field*, and how we propagate taint differs in these three cases to ensure overapproximation[4]. If a function is assigned to a function pointer variable, we simply add the variable into the tainted list for that function. Note that any type cast on function pointer variables is not a problem because our taint tracking approach simply propagates taint across assignments regardless of the data types of the variables.

Alternatively, if a function is assigned to any element in an array of function pointers, we taint the entire array. This is because programs normally access arrays using runtime indices. To be conservative, our approach assumes *f* could be retrieved from any index of the array.

Otherwise, if a function is assigned to a field in a structured type, we taint the field for all memory objects of that structure's type. That is, our *field-sensitive* overapproximation infers that if a function can be assigned to a field of one instance of a structure type, it can be accessed at any indirect call site that references any instance of that structure type. However, in theory, a program can access the function pointer field through a different structure type (e.g., unions). To recognize this case, we actively check for any alternative definition for a structured type, and taint the aliased function pointer field as well.

We detect violations of both assumptions while performing the taint tracking. Instead of resorting to an overapproximation, we report detected violations to the user and stop the analysis. To detect violations of A1, we actively check if any of the tainted memory objects are processed using an arithmetic operation. To detect violations of A2, upon every function pointer assignment, we check whether the function pointer, either on LHS or RHS, is accessed by directly dereferencing a data pointer. Note that this approach alone could miss the case where a function pointer is type cast to a different type such as `int`, `int*`, or even `void*`. We handle type casting in two ways. If a tainted memory object of a non-function-pointer type is accessed through pointer dereference, then it is a clear violation of A2. Otherwise, the tainted memory object must either be a local variable or a global variable, and the only way to create a pointer to a local or global variable is by explicitly using the `&` operator, which we detect as well.

5.2. Computing Return Targets

Return instructions are used in conjunction with call instructions. Thus, the key task in computing return targets are to map the call sites to their corresponding return instructions statically. This solution is straightforward for source code once the targets of all indirect call sites have been resolved. This is because source code has well-defined boundaries between functions, and we can easily identify the functions' return instructions. However, problems occur in assembly functions. First, programmers may apply certain compiler optimizations manually to assembly functions, which may hide the true return targets. Second, assembly functions may not adhere to the restrictions of functions in kernel source code, such as lacking well-defined boundaries, lacking return instructions altogether, and nesting functions inside of functions.

Tail-call optimization. Generally, a tail call is a function call that is performed as the final action of a procedure. One common optimization on tail calls is to reuse the current stack frame by deallocating local variables and *jump* to, as opposed to *call*, the target function, pretending it is being called by the caller of the current function. We show an example below. In this case, `cstart` is invoked by `main` and further calls `arch_init`. However, due to the optimization applied, `arch_init` will directly return to `main`.

```
main:            cstart:            arch_init:
  ...              ...                ...
  call cstart      jmp arch_init      ret
```

Fall-through functions. Assembly programmers may nest one function definition inside another, which enables the execution in one function fall through to the other. In the following example, `memset_fault_in_kernel` and `memset_fault` share the same function body except the former has additional handling. Therefore, the return instruction in the example may return to call sites that either invoke `memset_fault_in_kernel` or `memset_fault`.

```
memset_fault_in_kernel:
  ...
memset_fault:
  ...
  ret
```

In general, mapping returns to call sites is equivalent to finding the return instructions that are immediately dominated by the targets of every given call site. This problem can be solved using an intra-procedural control-flow analysis. Given a call site and its call targets, we analyze the CFG of the target to identify the immediately dominated return instructions, adding the instruction following the call site to the set of return targets for each return instruction.

Intuitively, given a call site, our approach emulates the execution of each target function and records the encountered return instructions. Starting from the target function, we sequentially "execute" the instructions one by one just as a processor does until reaching a control transfer instruction. If it is an unconditional jump, we follow its control flow, e.g., the tail call optimization case; otherwise (i.e., a conditional jump), we follow both edges. The search along any path stops "executing" under three conditions. First, the current instruction has been "executed". This is because any return instruction encountered later should have already been discovered. Second, the current instruction is a *stop* instruction such as `IRET`, `HLT` or a call instruction to a non-return function. This indicates the current function may not have a return. Third, the current instruction is a return

[4] Static taint tracking for assembly code must reason about registers and memory locations, which are represented as variables.

instruction. If this is the case, the call site is one of its targets. We summarize all the cases in Table 1.

Instruction	Action

`iret`
`hlt`
`sysexit`
`...`
	stop

`jmp`	**follow**
`jcc`	**continue** and **follow**
`call`	**continue** or **stop**
`ret`	**stop**
other	**continue**

TABLE 1: Cases in the emulated "execution"

The key challenge here is to identify non-return functions, which determines whether the "execution" should continue or stop on a call instruction. We solve this problem in a recursive way, that is, if the execution meets a call instruction, we look for its corresponding return instructions using the same procedure. As long as one return instruction is found, we continue the "execution"; otherwise, we stop. Note that recursive calls in programs can make this approach run into an infinite loop. Though we have not seen such case in assembly code, we propose to conservatively assume any recursive call will return eventually and let the "execution" continue. We run this approach on each call site to assembly functions, and combine the results from source code to produce a complete set of targets of function returns.

6. CFI Enforcement

In the second phase, we describe our approach to retrofitting the kernel software to enforce the CFG computed from the first phase comprehensively. We break this retrofitting effort into two tasks presented in this section and the next. In this section, we first outline in Section 6.1 an overview of the proposed approach for CFI enforcement over kernel software. Next, in Section 6.2 we detail the specific problems and solutions necessary to develop the proposed approach.

6.1. Approach Overview

Comprehensive CFI enforcement for kernel software is challenging because kernels must be capable of processing system events and have the power to configure how they execute such events and code in general. First, kernel software not only may perform indirect control transfers during their execution, but also may be entered by system events out of the kernels' control and exit either to run kernel or user-space code. Kernel entry can occur due to three types of events, system calls, interrupts, and exceptions, which may be triggered by user-space code or asynchronously via hardware. Second, unlike user-space programs, kernel software has much more control over how code is executed, including defining how memory segmentation is configured, which determines the effective address used for each instruction fetch, and configuring the code used for its event handling.

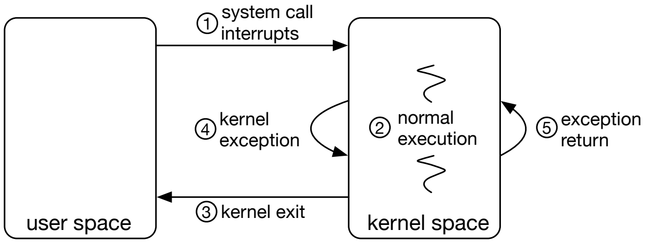

Figure 3: Execution model for non-preemptive kernels

An overview of an approach for comprehensive CFI enforcement in kernel software is shown in Figure 3. In step one, transitions from user-space to the kernel must only select known, approved entry points, such as event handlers for the three types of system events. To prevent an adversary from controlling kernel entry, we protect the integrity of various hardware-defined data structures that define those entry points. In step two, the kernel executes within the fine-grained CFG computed in the previous section. Then, two possible scenarios can occur. In one case shown in step three, the kernel completes an event handler and exits back to user space. Here, we must restrict the exit to run only at user-space privilege and restrict the target instruction run at the exit. These steps cover the execution of all system call events. Alternatively, in step four, a system event may occur during kernel execution. In this case, we must ensure that the event can be taken without disrupting CFI enforcement and that the exit from these events pick up where the kernel execution was. Unfortunately, we cannot guarantee such properties for interrupts that preempt kernel execution since they may occur at any time, so we instead integrate CFI enforcement into kernels built to run non-preemptively. We discuss the implications of this choice in Section 10. Kernel exceptions, on the other hand, either cause a kernel panic or occur at discrete times. The only type of exception we have found that needs special support are page fault exceptions. These only occur for specific kernel code (e.g., `copy_from_user`) and continue executing either from the instruction that caused the page fault or from a predefined location that handles invalid user pointers. Thus, step four only includes page fault exceptions, and in step five the exception's exit returns deterministically from the page fault handling. See Section 10 for further discussion of CFI enforcement over kernel exception handling.

Compared with existing CFI approaches for system software (i.e., HyperSafe and KCoFI), we restrict all the system events found in commodity operating system kernel while making enforcement more lightweight and efficient. For example, to ensure comprehensive control, we not only prevent memory writes to critical tables as HyperSafe does, but also remove instructions that would change the register value for these tables. As an example of improving performance, we relax the constraints on context switching: rather than ensure the kernel returns to the exact same user-space instruction, we only guarantee the privilege is lowered.

6.2. Enforcing CFI in Kernel Software

Specifically, to enforce the approach described above, we propose four system invariants.

- (**S1**) Segmentation configurations must be protected from adversaries.
- (**S2**) Execution must always start from a legitimate entry point.
- (**S3**) Throughout the execution, all indirect control transfers must adhere to the computed CFG.
- (**S4**) Exits must be mediated and authorized.

Enforcing S1. Kernel software poses additional risks to control-flow hijacking attacks when compared to user-space applications because of the use of architecture-defined control data for various system configurations. Among these, memory segmentation (i.e., the Global Descriptor Table, or GDT) is fundamental in the sense that the code segment determines the base address used by both direct and indirect control transfers. We protect the GDT based on the observation that most operating system kernels never modify it once set. Therefore, we (1) zero out the `LGDT` instruction to prevent the adversary from setting up her own GDT via code reuse, and (2) write-protect the GDT throughout the system lifetime to prevent any in-place modification. To eliminate the side effects of the changed memory permissions, we separate the GDT from other kernel data by moving it into a distinct page.

Enforcing S2. Kernels are triggered by system events such as interrupts, exceptions and system calls. Depending on the specific event, the kernel will invoke different routines to handle the event. Upon interrupts, the control will be transferred to the routine specified in the Interrupt Descriptor Table (IDT) defined by the kernel. Alternatively, a system call will switch the current execution to the kernel at the address defined by two Model Specific Registers (MSRs) `IA32_SYSENTER_CS:IA32_SYSENTER_EIP`[5]. To satisfy S2, we must guarantee all these system events will trigger the kernel execution from legitimate entry points defined by the kernel. We achieve this goal in two steps. First, similar to GDT protection, we require both removal of `LIDT` instruction and write-protection for the IDT throughout the system lifetime. This prevents the adversary from manipulating the entries to interrupt handling. Second, we require that MSRs related to system calls are never modified once set. We achieve the latter by zeroing out all `WRMSR` instructions in the kernel after the system is booted. Note that this may not apply to arbitrary kernels because some do need to modify MSRs after boot. For these cases, we propose to duplicate the `WRMSR` instructions and hardcode the register identifier for them to guarantee that it can only write to MSRs other than the two related to the system call entry. The protection of both interrupt and system call entries satisfies S2.

Enforcing S3. To hijack the control flow during kernel execution, the adversary has to either control the return of a kernel exception or an instrumented indirect control transfer. To protect the return of page fault handling from adversaries, we only allow the kernel to return either to the predefined location that handles invalid user pointers or to the exact instruction that took the page fault. To achieve the latter, we store the return address to an unused debug register (or anywhere that is free of memory corruption) at the very beginning of exception handling, and check the subsequent exception return against the address we saved before. To prevent the adversary from controlling an instrumented indirect control transfer, we write-protect the target tables used by restricted pointer indexing. To eliminate the *time-of-check to time-of-use* (TOCTTOU) attacks on the index, in addition to disabling interrupts, we load the index into registers before executing the check. This approach not only guarantees no exception could ever happen in between, but is also safe from inter-processor races. Thus, S3 is satisfied.

Enforcing S4. The kernel will return the control to the user space eventually. Conceptually, the kernel returns the control back to the user space upon the completion of an interrupt (e.g., context switch) or a system call. In both cases, since the target instructions are outside of the kernel's CFG on purpose, we do not apply the same CFI enforcement to them. Instead, in general, we ensure that these indirect control transfers would switch the subsequent execution to the user privilege (i.e., ring 3 on x86), preventing the adversary from launching control-flow hijacking attacks back to the kernel space. However, for system call return, we apply a stricter enforcement based on the fact that, for portability reasons, most kernels map the routine for invoking system calls to the user-space address space so that programs do not have to concern themselves with whether a system call is implemented using the software interrupt, `SYSENTER` or even its equivalent instruction `SYSCALL` on AMD processors. This implies that programs that are linked with standard library should all obey this convention and make system calls through the kernel-mapped routine. Therefore, we add checks on the return address of `SYSEXIT` (stored in `EDX`) against the address of the `SYSCALL`-preceded instruction before returning to the user space, and panic the system if they do not match. Note that this will potentially break the programs that make system calls through hand-written software interrupts. However, we did not encounter any such case in practice.

7. CFI Instrumentation

In this section, we describe our proposed approach for instrumenting kernel software to enforce the fine-grained CFG computed in Section 5. The idea is that we start with an effective, general instrumentation approach and when we detect a possible optimization opportunity we replace the

[5] Traditionally, system calls are implemented using software interrupts (i.e., `INT`). Later, in order to enable faster transition, Intel introduced a pair of dedicated instructions `SYSENTER` and `SYSEXIT` for this purpose. MINIX supports this mechanism while FreeBSD still uses software interrupts for 32-bit kernels.

default instrumentation with optimized instrumentation. We first in Section 7.1 briefly introduce the concept of restricted pointer indexing. Second, in Section 7.2 we describe two opportunities for optimizing CFI instrumentation. Specifically, we leverage the fine-grained CFG to replace instrumentation with direct control transfers when there is only one legal target, and we also make use of checks already in the kernel code to further optimize the instrumentation.

7.1. Restricted Pointer Indexing

Restricted pointer indexing is a CFI enforcement technique proposed by Wang *et al.* [44]. We choose this technique as our default for CFI instrumentation because it avoids the destination equivalence issue, which limits the granularity a CFI defense could achieve, by explicitly separating target sets for each indirect control transfer. Its main idea is to organize all the control data, e.g., return addresses and function addresses, into target tables and replace all the addresses used in indirect control transfers, either statically initialized (e.g., function pointers) or dynamically generated (e.g., return addresses), with corresponding indices.

To use restricted pointer indexing, one must transform all the indirect control transfers in such a way that they all use indices for transferring control by fetching the target address from the target tables. Note that for control transfer targets that may appear in multiple tables, its index has to be unique among all target tables. This is because when an index is introduced, e.g., by a call instruction, it can be used at multiple destinations, e.g., return instructions. Hence, if an index is mapped to different targets when used by different indirect control transfers, it would make the transformed program behave inconsistently. To ensure the same target must have the same index in all tables, we use padding entries when necessary. The padding entries point to the *panic* function in case they are misused by adversaries.

7.2. Optimizing CFI Enforcement

One key obstacle that prevents CFI from receiving wide adoption is the performance cost. We identify two opportunities to reduce the amount of instrumentation required for CFI enforcement.

Our first observation is that compilers may already insert code to restrict the targets of indirect control transfers to safe values, so we can simply leverage this available instrumentation. For instance, all indirect jumps produced when compiling *switch* statements use a read-only jump table and an index value for choosing the *case* statement to be executed. The compiler adds code to check that the index value is within jump table, so this code satisfies CFI by default. Similarly, in kernel software, programmers store references to all the system call service routines in a table and invoke the requested routine based on the system call number (i.e., index). The system call number (index) is restricted to be within the table. Given the high frequency of system call requests from user space, adding code to enforce this table index redundantly would lead to an unnecessary performance impact. If this code complies with instrumentation of *base plus index* cases shown in Table 2, then we do not need to add bounds checks[6]. Finally, kernel programmers may leverage absolute addressing for making direct control transfers in assembly code. In this case, they hardcode a jump target into a register, and then immediately make a control transfer. We claim that no enforcement is required for these indirect control transfers because they have a deterministic target at runtime.

Our second observation is that many kernel indirect control transfers have fixed targets that can be identified at link time. To enable flexible kernel configurations, kernel programmers often use function pointers that can be bound to the particular modules' functions specified in the kernel configuration. In many cases, the configuration specifies only one target for these function pointers. For example, in the MINIX microkernel, the kernel uses the ELF library to load server binaries. The library is designed to be generic so it allows callers to specify different functions to perform core tasks, e.g., memory allocation. However, the microkernel is assigned a fixed set of these functions at link time, so those function pointers have fixed values. Surprisingly, we found that the percentage of indirect calls which have fixed targets ranges from 31% to 66% across the FreeBSD kernel, MINIX microkernel and its user-space servers (see Section 9 for details). Therefore, we simply rewrite these indirect control transfers to semantically-equivalent direct control transfers to minimize both memory and runtime overheads.

8. Implementation

We implement our techniques on both FreeBSD 10.0 and MINIX 3.2.1 for Intel x86 platforms in four parts. The first part is to change the build process of kernel software so that kernel object files are linked against recompiled libraries rather than the system libraries because our technique requires whole-program analysis and instrumentation. We achieve this part by instrumenting the static linker to identify all relocatable object files that make up the final executable.

The second part is to compute a fine-grained CFG using the techniques presented in Section 5. We implemented this part as an LLVM pass. It contains 1,391 lines of C++ source code and runs on LLVM 3.5 [28]. The pass takes LLVM bitcode as input and returns the target functions of each indirect call site. Throughout its analysis, the pass will actively evaluate if any assumption made in Section 5.1 has been violated and raise an exception upon detection. For indirect calls and jumps written in assembly code, we detect them, but have to analyze them manually. However, we find that most indirect calls and jumps in the assembly code comply with the code pattern described in Section 7.2 thus are free of control-flow hijacking attacks.

The third part is to modify the kernel to enforce system invariants presented in Section 6.2. We set up the protection,

[6] However, we add code to store the index for call instructions that use *base plus index* addressing to identify the return target (caller).

including removing various instructions and write-protecting configuration data structures at the end of system booting but before the kernel initiates the first user-space process. This allows the booting procedure to freely configure them without causing false positives. To eliminate the side effects of the changed page permission, we also relocate both GDT and IDT so that they are page-aligned and not co-located with other kernel data. To protect the kernel exiting to user space, we check the code segment selector that is about to be restored, and ensure that it points to the user code segment defined in the GDT.

The last part is CFI instrumentation. This part has two steps. The first step is to instrument each relocatable object file individually, including replacing control data with indices, rewriting indirect calls and returns to equivalent index-aware instructions as designed by restricted pointer indexing. We summarize our instrumentations[7] in Table 2. It is worth mentioning that, for function calls, in addition to specifying the return address for the callee, our instrumentation pushes the return index before checking the target to inform the *panic* function where things went wrong. At this step, indices, base addresses of target tables, and their contents are not determined yet because actual addresses of targets are only known after linking. To help the next step recognize the original indirect control transfers (e.g., indirect call sites and return instructions) we use a magic number `0xdeadbeef` for these unknown values. The second step is to fill each target table and update indices that are left blank in the first step. This step identifies the original call sites and return instructions via the magic number and updates them to the correct value. We implement both steps of instrumentation in a Python script containing 526 lines of source code.

There are a few cases we experienced where automatically applying the CFI instrumentations can break a system. For instance, in the context switch routine, the FreeBSD kernel uses a return instruction to resume the execution of the scheduled thread. However, newly created threads do not have a valid index for the instrumented return to function properly. Therefore, we manually handle the return by adding the corresponding index into its target table. Also, the FreeBSD boot process reloads the code segment register after paging is enabled via intersegmental return[8]. Since the return address is substituted with an index, it generates a fault. To make this instrumentation work, we manually rewrite the routine so that it uses `LJMP` to reload the code segment register, and uses a normal `RET` to return to the caller. Similarly, when retrofitting MINIX servers, we found an assembly library function `getcontext()` that attempts to directly read the return address pushed by a call site, which crashes the servers. We handled this issue by manually rewriting the procedure to be index-aware. In addition, in the MINIX microkernel, the system call invocation routine that is mapped to user space must not be instrumented because (CFI-unaware) user-space programs expect an address rather than an index. Since this routine is only executed by user-space programs, we can leave it as is but mark it as non-executable from the kernel space via Intel's Supervisor Mode Execution Prevention (SMEP) feature [3] to prevent attacks.

9. Evaluation

In this section, we evaluate our techniques from three perspectives: technique utility, security improvement, and performance overhead. We first show the results of our analysis proposed in Section 5.1 on a variety of kernel software, including the FreeBSD kernel, the entire MINIX microkernel system and the BitVisor hypervisor. Next, we quantitatively evaluate the improvement on CFI protection achieved by this work. Finally, we evaluate the performance cost of our fully-implemented, fine-grained CFI enforcement on FreeBSD 10.0 and the MINIX 3.2.1 systems.

9.1. Technique Utility

We evaluate the utility of the proposed analysis on three different kernel systems (e.g., FreeBSD, MINIX, and BitVisor) by counting the source lines of code (SLoC) using the open-source tool `SLOCCount` [2] against the number of constraint violations encountered. Note that when counting the SLoC of MINIX, we do not include library code that is linked to the final executable, but list them separately.

We show our results in Table 3. We have seen only one violation in BitVisor, where some code uses a constant pointer to a global function pointer for saving a replaced handler. We manually rewrite the code so that it directly references the global function pointer. In FreeBSD, six of the seven violations are in device drivers. Surprisingly, we found that one violation is because of a driver purposely overflowing structure fields in assignment operations. For the ease of analysis, we opt to exclude these drivers from the kernel image, as they are not required by our computer's configuration, rather than to manually fix them. However, the other violation is in the core part of the FreeBSD kernel, which manages a special group of function pointers on the heap. Fortunately, the used data pointers neither point to a structure field nor to any function pointer variables, therefore, we manually identify those targets and add them to the computed CFG.

9.2. Security Evaluation

In the security evaluation, we answer: (1) how much stricter our enforcement of CFI is relative to existing CFI implementations and (2) how many gadgets remain. To quantitatively measure the improvement, we proposed a new

[7] We use `ECX` to hold the index of return address and `EAX` to hold the call target if the original call site references memory because both `EAX` and `ECX` are caller-saved registers in most calling conventions.

[8] In addition to changing the instruction pointer like a normal return, `LRET` instruction also updates the code segment register with a 16-bit segment selector from the stack.

[9] BitVisor is a Type-1 hypervisor and we only test our techniques on its core part.

Addressing mode	Original code	Basic instrumentation	Fixed-target instrumentation
Register Indirect	call eax	push index cmp eax,bound ja panic jmp [table+eax*4]	push index jmp foo
Memory Indirect	call [ebp+offset]	push index mov eax,[ebp+offset] cmp eax,bound ja panic jmp [table+eax*4]	push index jmp foo
Memory Indirect	ret	pop ecx cmp ecx,bound ja panic jmp [table+ecx*4]	pop ecx jmp target
Base plus Index	cmp eax,bound ja some_label call [table+eax*4]	push index cmp eax,bound ja some_label jmp [table+eax*4]	N/A
Base plus Index	cmp eax,bound ja some_label jmp [table+eax*4]	N/A	N/A

TABLE 2: CFI Instrumentation based on Original Code and Addressing Mode

	SLoC	A1	A2
FreeBSD	9,280,785	0	7
MINIX	16,276	0	0
vfs	9,783	0	0
vm	6,633	0	0
pm	2,856	0	0
rs	3,221	0	0
ds	587	0	0
pfs	2,625	0	0
sched	393	0	0
libc	153,264	0	0
libsys	4,336	0	0
libmthread	1,532	0	0
BitVisor[9]	34,987	0	1
total	9,517,278	0	8

TABLE 3: The amount of code (in SLoC) being analyzed *vs.* the number of violations detected

metric, called *Average Indirect targets Allowed* (AIA). In addition, we also show the distribution of number of allowed targets per indirect control transfer (calls and returns) in the FreeBSD kernel, MINIX microkernel, and two of its crucial user-space servers. To answer the second question, we analyze the ROP gadgets remaining in the fine-grained CFI-enforced FreeBSD and MINIX using the open-source tool ROPgadget [36].

Researchers previously proposed a metric to evaluate the effectiveness of CFI enforcement, called *Average Indirect target Reduction* (AIR), proposed by Zhang *et al.* [46]. The problem with AIR is that it does not provide an intuition about how much a CFI implementation over-approximates the CFG, especially when the code base of the protected program is large. Imagine an x86 program with 1MB code. Given that every byte could potentially lead an instruction, a CFI implementation achieving AIR of 99% still allows 10K targets for each indirect control transfer on average. However, in practice, an indirect control transfer often has many fewer targets (e.g., <10). Moreover, recent attacks [12, 19, 23] on coarse-grained CFI implementations show that an adversary often needs only a small subset of unintended control flows to realize her purpose. Fine-grained CFI can also be attacked if the target sets allow an attacker to fulfill her needs [11].

In order to better illustrate the protection quality, we propose the *Average Indirect targets Allowed* (AIA) metric, as defined in Definition 1. It is worth noting that this metric can only be used to compare CFI techniques on the same program as different programs can be constructed differently, yielding vastly different results.

Definition 1 (Average Indirect targets Allowed (AIA)). *Let I_1, I_2, ..., I_n be indirect control transfer instructions in the program, and T_1, T_2, ..., T_n be the set of allowed targets for them respectively. We define*

$$AIA = \frac{1}{n}\sum_{i=1}^{n} |T_i|.$$

To evaluate, we assume a coarse-grained CFI policy that uses two target sets as adopted by some recent work [17, 46] and a signature-based CFI policy as used elsewhere [30, 41]. We show the results in Table 4. From this comparison, our techniques enforce a much tighter policy than a traditional CFI implementation as it further reduces over 99.6% return targets and over 99.1% call/jump targets that are otherwise allowed. Our approach also has higher precision than signature-based CFI as it reduces between 71.80% and 94.03% of call/jmp targets across the three different kernels. Signature-based approach performs slightly better

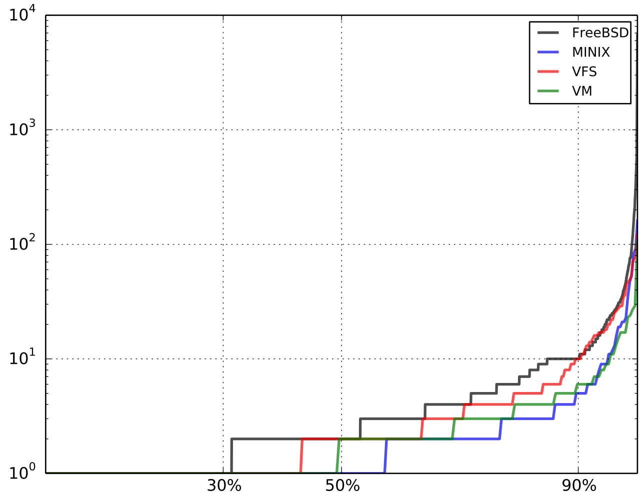

Figure 4: Distribution of the number of indirect control transfer targets in the computed CFG from Section 5

on MINIX servers because they have fewer address-taken functions and less assembly code, but still is between 37.6% and 49.8% worse than our proposed approach. In addition, we also find signature-based CFI will break both the FreeBSD and MINIX system due to missed targets, such as the one caused by signature cast, e.g., `int (*)(void)` to `int (*)(struct irq_hook *)`.

Next, we show the distribution of the number of allowed targets per indirect control transfer for FreeBSD, MINIX microkernel and its two biggest user-space servers in Figure 4. In summary, over 30% of indirect control transfers have fixed targets and over 90% have fewer than ten targets. However, in both FreeBSD and MINIX, there are a few indirect control transfers that have a large number of allowed targets, e.g., the return of `printf` function in FreeBSD has over 5K allowed targets. Note that, although every allowed target corresponds to a valid edge in the CFG, such flexibility could make the indirect control transfer an attractive target to adversaries and enable adversaries to find potentially useful code for exploitation. Though the original CFI proposal [6] has suggested the use of a shadow stack for more restrictive protection, how to efficiently implement a shadow stack in system software remains an open problem. Safe stack [1, 27] has better performance than shadow stack, but it changes the stack layout and hence has compatibility issues when applied to kernel software. We will explore optimizations to these approaches as future work.

Finally, to measure the impact on code-reuse attacks from the perspective of reusable code, we analyze the ROP gadgets remaining in the instrumented systems using the open-source tool ROPgadget version 5.2. ROPgadget finds 61565 gadgets in the original FreeBSD kernel and 2164 gadgets in the original MINIX. The numbers reduced to 388/30 after our instrumentation. We verified all control transfers (both *direct* and *indirect*) allowed by the CFG and confirmed that none of these gadgets could be reached through a control flow we authorize. However, recent work [12, 19, 23] has shown that the adversary can use more complexed code sequences than traditional gadgets for attacks, thus we provide this analysis as a lower bound of reusable code.

9.3. Performance Evaluation

In this section, we evaluate the performance overhead of our fine-grained CFI implementation on FreeBSD 10.0 and MINIX 3.2.1 including all of its user-space servers. Our test machine consists of a dual-core processor running at 2.8GHz with 1GB memory. We use the unmodified systems as the baseline and evaluate the modified versions that are protected by both the traditional coarse-grained CFI and our fine-grained CFI techniques.

Before presenting the actual results on performance overhead, we first show the distribution of different instrumentations in Table 5. These statistics indicate that each of the three types of indirect control transfers are well represented in real world programs, so we need to apply specialized instrumentation to them to save both runtime and memory overheads. For example, indexing the system call table is an example of a base-plus-index indirect control transfer that requires no additional enforcement. Adding extra instrumentation to unify the implementation of restricted pointer indexing or even inserting a label at system call handlers as proposed by original proposal is unnecessary, given the frequency that this instruction is executed.

To evaluate the performance, we run the UnixBench 5.1.2 as the microbenchmark and build the kernel as the macrobenchmark. We chose UnixBench because it runs both on FreeBSD and MINIX. We show the runtime performance overhead in Figure 5. In MINIX, the fine-grained CFI instrumentation incurs only 2.02%/5.64% (average/-maximum) overhead while the coarse-grained CFI instrumentation incurs 4.14%/7.55% overhead. In FreeBSD, the overhead is higher but the fine-grained CFI still performs better than the coarse-grained CFI (i.e., 11.91%/42.03% vs. 13.60%/50.69%). We attribute much of the performance difference between FreeBSD and MINIX to the cache performance: a monolithic kernel like FreeBSD has one large set of target tables while in a microkernel, each component has its own set of target tables, leading to better locality. We will explore how to colocate these tables in FreeBSD to improve the cache performance as a future optimization. For the macrobenchmark, the fine-grained CFI uses 1.82%/0.76% more time while coarse-grained CFI uses 2.01%/2.29% more time to build the FreeBSD/MINIX system. We believe that enforcing fine-grained CFI can be as efficient as existing coarse-grained CFI implementations.

Enforcing CFI uses more memory. On one hand, the instrumentation will require more instructions for the authorization of indirect control transfers. On the other hand, restricted pointer indexing requires saving all control data into target tables, which are absent from the original system. On average, the typical coarse-grained CFI uses 25.3% more code memory. However, the fine-grained CFI uses 13.48% less static data memory and 9.01% less code memory than

	Fine-grained CFI		Signature-based CFI		Coarse-grained CFI	
	`call/jmp`	`ret`	`call/jmp`	`ret`	`call/jmp`	`ret`
FreeBSD	6.64	10.41	35.61	11.38	40,166	175,400
MINIX	4.67	5.14	16.56	9.29	688	2,008
vfs	2.89	5.11	4.63	5.38	578	2,289
vm	2.14	4.01	3.43	4.39	402	1,149
pm	2.55	2.93	4.88	5.38	281	608
rs	1.68	4.61	3.13	4.94	354	1,100
ds	1.67	3.85	2.89	4.13	201	561
pfs	2.54	4.95	4.42	5.43	912	2,785
sched	1.23	2.62	2.45	2.91	197	333
BitVisor	3.84	1.89	64.27	6.55	911	3,864

TABLE 4: AIA of indirect control transfers in kernel softwares

	Basic	Fixed-target	Unchanged
FreeBSD	13,143	6,019	1,207
MINIX	138	199	11
vfs	269	184	22
vm	168	154	3
pm	89	165	5
rs	128	161	2
ds	75	101	3
pfs	360	307	20
sched	53	104	1

TABLE 5: Distribution of different instrumentations applied

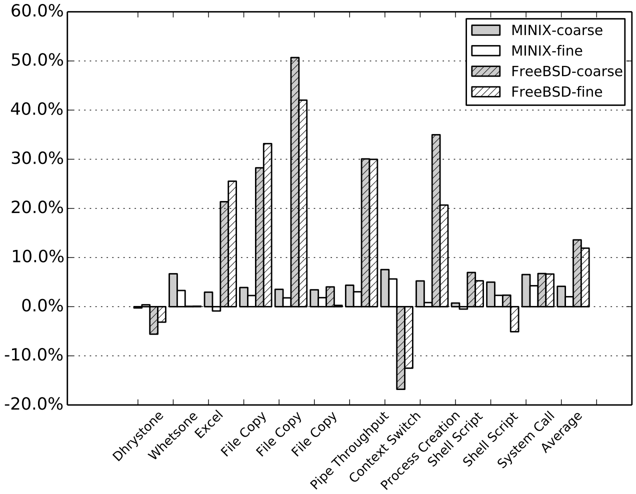

Figure 5: Overheads of running UnixBench 5.1.2 on both coarse-grained CFI-instrumented and fine-grained CFI-instrumented FreeBSD and MINIX

the coarse-grained CFI. There are two reasons. First, the fixed-target instrumentation saves both code memory, i.e., it has fewer instructions, and data memory, i.e., it does not need a target table. Second, unlike coarse-grained CFI, not all call sites necessarily have an entry in the target table because some call sites are not mapped to any return instruction. For instance, the function to restore user context has an `IRET` instruction instead. Therefore, even though fine-grained CFI may have redundant targets in the tables, it has better memory use than the coarse-grained version in our experiment.

10. Discussion

Kernel preemption is a technique to increase the responsiveness of the overall system. Basically, it allows interrupting kernel execution (e.g., system call handling) to schedule a higher-priority task onto the processor. This affects the kernel control flow in an unpredictable way, as the kernel execution can be interrupted and returned to any instruction boundary. In this work, we build FreeBSD as non-preemptive[10] to avoid this issue. Although a preemptive kernel may increase the system responsiveness, it could degrade the system performance because of increased context switching. However, a kernel exception could only happen at certain places, and its handling and return are deterministic. FreeBSD uses kernel exceptions to emulate the virtual 8086 execution, which purposely modifies the exception return address. To avoid increasing the complexity of our implementation, we simply disable this feature as it is not required in our kernel.

Although CFI provides the first line of defense to prevent the adversary from subverting the MMU configuration, memory protection is required in order to protect our CFI enforcement from data attacks. While earlier work on CFI enforcement in kernel software included memory protection mechanisms, they either left a window for attack (turn on and off memory protect in HyperSafe [44]) or were expensive (virtual machine support for KCoFI [17]). However, recent research has shown that that protecting MMU can be done efficiently (2.5% reported by Readactor [16]) and comprehensively. Unfortunately, these solutions either cannot be directly applied to FreeBSD/MINIX kernels [22, 34] as used in this work, or not publicly available yet [16], so we assume the presence of such protection rather than implementing

[10] MINIX cannot be built as preemptive by design. In fact, many operating systems do not enable kernel preemption by default (e.g., Linux).

our own. However, these techniques used are orthogonal to the CFI techniques discussed in this paper, so we envision adding such memory protection as future work.

Control-Flow Bending [11] combines data attacks with control-flow hijacking attacks to enable malicious computation and showed the requirement of a stack integrity to protect user-space applications. As mentioned in Section 3, we do not protect the kernel from data attacks. However, our fine-grained CFI policy makes the control-flow bending attacks less likely to succeed because of the smaller size of targets allowed at runtime. Unfortunately, we found that kernel software includes some cases with large target sets, i.e., the return of functions that are commonly called, so these may also be prone to control-flow bending attacks. The authors of that work recommend applying the shadow stack (or safe stack) approaches to prevent such attacks, but no kernel implementation of that technique is available yet. Kernel implementations of stack integrity is challenging as exception handling and other instructions (e.g., `iret`) assumes a specific rigid stack layout that cannot easily be changed. We will explore implementing such mechanisms in kernel software as future work.

11. Related Work

In 2007, Shacham *et al.* introduced an advanced code-reuse attack vector called return-oriented programming [38]. Later researchers extended ROP attacks to more platforms such as ARM [26] and other RISC architectures [10]. They found the attack is Turing-complete, making it a generally applicable threat to systems protected by DEP. Based on that, Hund *et al.* created a return-oriented rookit [24] to bypass IDS monitors [34, 37]. Similarly, Vogl *et al.* leveraged ROP to implement dynamic hooks [42] and persistent data-only malware [43] to avoid using explicit hooks or introducing new code, making its detections harder.

There were some initial attempts to defend against ROP attacks. Li *et al.* built a kernel without return instructions to defeat the attack [29]. However, both [14] and [9] demonstrate that code-reuse attacks can be launched without using return instructions. These works demonstrate that all types of indirect control transfers are potential targets for adversaries.

Abadi *et al.* first introduced the concept of control-flow integrity [6]. Researchers propose to use CFI to mitigate code-reuse attacks because they often need to divert the original program's control flow to perform desired computation.

Previous work mainly focused on how to make CFI more practical. Zhang *et al.* presented a binary-rewriting approach to enforce CFI on COTS binaries comprehensively [46]. Depending on relocatable information available on most Windows applications, CCFIR [45] has more reliable disassembly and static rewriting, and it also enforces CFI on Windows binaries more efficiently than prior work [46]. At the compiler level, modular control-flow integrity (MCFI) added separate compilation supports for CFI [30], which allows modules to be independently instrumented and linked.

Researchers also proposed approaches to enforce lightweight CFI without modifying programs at all. kBouncer uses branch tracing hardware available on modern processors to detect ROP attacks [32]. It is efficient as it only checks the branch histories upon system calls and kills the program if an ROP attack is detected. ROPecker builds on the ideas of kBouncer [15]. However, it checks more often and thoroughly than kBouncer.

People have also examined privileged system software. HyperSafe enforces lifetime CFI for a hypervisor [44]. We adopted their instrumentation approach (i.e., restricted pointer indexing) in this work. kGuard protects the kernel from *ret2usr* attacks by enforcing all indirect control transfers within the kernel must not target a user-space address [25]. Criswell *et al.* made the first attempt to enforce CFI on conventional operating systems [17]. They built their enforcement on top of the secure virtual architecture [18], a software layer between the operating system and hardware. However, their implementation suffers from high performance overheads.

There are some recent efforts on tightening the enforced CFI policy. Forward-edge CFI [41] uses a signature-based approach to reduce the target set of indirect calls. Modular CFI [30] uses a similar approach to enforce a stricter CFI policy. However, signature-based approach has both false positives and false negatives on our tested kernel software. Per-input CFI [31] takes one step further by incrementally building the enforced CFG at runtime, thus is able to reject control transfers that are within the program's CFG while invalid for the current concrete input. However, Per-input CFI requires secure runtime code patching, which is challenging for kernel software. We will explore this direction as future work.

Unlike CFI that prevents programs from using unauthorized control data, code-pointer integrity ensures that adversary must not overwrite the control data in the first place [27]. They modified the compiler in such a way that sensitive data such as return addresses are stored in a safe region that adversaries have no easy access to.

However, a few recent proposals have shown that all the coarse-grained CFI implementations are insufficient to block advanced ROP attacks. Göktaş *et al.* demonstrated an attack on Internet Explorer 8 against coarse-grained CFI defenses [23]. Davi *et al.* also showed a similar attack against CFI defenses [19]. Carlini *et al.* proposed history-flushing approaches to bypass branch tracing as leveraged by kBouncer and ROPecker [12]. Control-Flow Bending [11] highlights limitations of CFI as a defense mechanism by itself. Our fine-grained CFI-protected kernel significantly raises the bar for these attacks in practice, making them less likely or even impossible.

12. Conclusion

In this paper, we have presented an automated approach for enforcing fine-grained control-flow integrity for kernel software. We leverage and enforce constraints kernel software adheres to for function pointers to compute a fine-grained control-flow graph and authorize the indirect control transfers in the resulting graph comprehensively and efficiently. We also address system challenges that arise only in kernel software to protect its entries and exits. The result shows that our approach is both effective, i.e., eliminating over 70% of the indirect call targets in the FreeBSD kernel and MINIX microkernel that are otherwise allowed by current fine-grained CFI implementations and 99% of the indirect control targets in a typical coarse-grained CFI implementation, and efficient, i.e., incurring less overhead than a comparable coarse-grained CFI implementation.

13. Acknowledgement

We thank the reviewers for their constructive suggestions to this work. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). This material is based upon work supported by the National Science Foundation under Grant No. CNS-1408880 and No. CNS-1513783. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] LLVM SafeStack. http://clang.llvm.org/docs/SafeStack.html.

[2] SLOCCount. http://www.dwheeler.com/sloccount/.

[3] Supervisor mode execution protection. http://vulnfactory.org/blog/smep/.

[4] Intel IA-32 architectures software developers manual. *Volume 3b: System Programming Guide (Part 2)*, pages 14–19, 2013.

[5] Linux kernel vulnerabilities. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2015.

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

[7] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 263–277, 2008.

[8] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

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

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *SEC'15: 24th Usenix Security Symposium*, 2015.

[12] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, Aug. 2014. USENIX Association.

[13] M. Castro, M. Costa, and T. L. Harris. Securing software by enforcing data-flow integrity. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 147–160, 2006.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

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

[16] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy (S&P 2015), 18-20 May 2015, San Jose, California, USA*, 2015.

[17] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 292–307. IEEE, 2014.

[18] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 351–366. ACM, 2007.

[19] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association.

[20] I. Fratrić. ROPGuard: Runtime prevention of return-oriented programming attacks. 2012.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 193–206, 2003.

[22] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. In *Proceedings of the 3rd IEEE Mobile Security*

Technologies Workshop (MoST 2014), May 2014.

[23] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[24] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[25] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium*, pages 459–474, 2012.

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

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

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[29] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[30] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 58. ACM, 2014.

[31] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914–926. ACM, 2015.

[32] V. Pappas. kbouncer: Efficient and transparent ROP mitigation. Technical report, Citeseer, 2012.

[33] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *DIMVA'15: 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2015.

[34] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2008.

[35] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[36] J. Salwan. ROPGadget. http://www.shell-storm.org/project/ROPgadget/.

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[38] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[39] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, pages 72–89, 2002.

[40] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[41] 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 *USENIX Security Symposium*, 2014.

[42] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic hooks: hiding control flow changes within non-control data. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 813–828. USENIX Association, 2014.

[43] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[44] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[45] 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 *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[46] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, pages 337–352, 2013.