

DROP THE ROP

Fine-grained Control-flow Integrity for the Linux Kernel

João Moreira
jmoreira@ic.unicamp.br
Computing Institute
University of Campinas

Sandro Rigo
sandro@ic.unicamp.br
Computing Institute
University of Campinas

Michalis Polychronakis
mikepo@cs.stonybrook.edu
Dept. of Computer Science
Stony Brook University

Vasileios P. Kemerlis
vpk@cs.brown.edu
Dept. of Computer Science
Brown University

Abstract

The introduction of W^X memory policies and the subsequent mitigation of return-to-user attacks, has rendered return-oriented programming (ROP) the most prominent exploitation method of kernel-level vulnerabilities. Control-flow integrity (CFI) is an effective defense against ROP, but despite its many refinements during the past decade and its recent deployment for the protection of user-space applications, it has received significantly less attention for the kernel setting. The few existing kernel-level CFI proposals either apply an overly permissible coarse-grained policy or do not support dynamically loadable kernel modules, making their deployment impractical for Linux. In this paper we present the design and implementation of kCFI, a fine-grained CFI implementation for commodity operating systems (OS) that fully supports the Linux kernel. By combining static analysis at both the source code and binary level, kCFI generates a more restrictive CFI policy compared to previous proposals that is enforced through the addition of control flow checks during compilation. The side-effects of kCFI are minimal, as it does not harm any OS functionality or feature, and achieves a lower overhead compared to previous solutions, in the order of 8% and 2% for micro and macro benchmarks, respectively.

1 Introduction

From a security perspective, the operating system kernel is a high-value asset, as it operates under a privileged mode that allows ambient access to every system resource. The exploitation of kernel-level software vulnerabilities has thus been a major goal of attackers for achieving privileged full system access. In the past few years, kernel exploits have seen renowned interest, as the exploitation of user-space client and server applications is becoming more challenging due to the deployment of sandboxing and container technologies, along with numerous other exploit mitigations. Once confined within a sandbox, it is often easier for an attacker to exploit a kernel vulnerability to gain full system access, instead of finding a sandbox-specific vulnerability. The former is typically easier due to the abundance of exploitable kernel bugs, and the relative lack of kernel-level exploit mitigations compared to user space programs. Indicatively, already before the end of 2016 there have been 427 reported kernel vulnerabilities according to the National Vulnerability Database, 172 more than in 2015 [71].

Due to the complexity and unique characteristics of the kernel, the deployment of exploit mitigations is often lacking or is not existent at all compared to user space. For instance, although address space layout randomization (ASLR) [78] has started being employed by major Operating System (OS) kernels [30], it still suffers from limited entropy issues compared to user-space implementations. In addition, even when there are no entropy issues, kernel memory leakage or side channel vulnerabilities can be leveraged by attackers to pinpoint module addresses and bypass ASLR [95, 96]. Given that return-oriented programming (ROP) [94] is becoming the most prevalent exploitation technique at the kernel setting, this is a crucial issue, as ASLR is currently the main deployed mitigation against kernel-level code reuse attacks in modern OS kernels.

Control-flow integrity [8] is an additional, orthogonal defense against ROP attacks that after many refinements in academic research [10, 16, 27, 73, 74, 77, 82, 84, 105, 114–116] is finally getting traction in user space with compiler, OS, and hardware support [1, 2, 7, 105]. By confining program execution within the bounds of a precomputed profile of allowed control flow paths, CFI can prevent most of the irregular control flow transfers that take place during the execution of ROP code.

In contrast to the large body of work on CFI for the protection of user-space programs, only a few efforts have focused on the application of CFI at the kernel setting [24, 38, 57, 101]. Due to the complexity of kernel code and

its unpredictable execution behavior caused by interrupts and exceptions, existing implementations either apply an overly permissible coarse-grained policy, to avoid the complexity of extracting a complete control flow graph [24], or are not compatible with dynamically loadable kernel modules, to facilitate the extraction of the control flow graph (CFG) needed for deriving a more fine-grained policy [38]. Coarse-grained CFI offers only limited protection, as it still permits plenty of valid code paths for the construction of functional ROP exploits [14, 28, 39, 40, 92], while lack of support for dynamically loadable modules limits the practical applicability of the protection, especially for Linux, which relies heavily on them.

As a step towards practical and effective kernel-level control-flow integrity for commodity operating systems, in this paper we present kCFI, a fine-grained CFI scheme for the Linux kernel. The proposed approach combines the benefits of a tag-based, fine-grained CFI policy enforcement for both forward and backward edges which offers increased protection compared to coarse-grained CFI, with full support of dynamically loadable kernel modules, a crucial feature for supporting the Linux kernel that is missing in some previous kernel-level CFI solutions. kCFI is a purely compiler-based approach, and its CFI enforcement mechanism does not rely on any runtime supervisor module or routine, avoiding any associated overheads.

To automatically generate the enforced policy, kCFI extracts the kernel’s CFG by statically analyzing its code at both the source code and binary level. This approach captures a detailed view of the CFG that implicitly deals with challenges such as aliasing or divergence between final machine code and its high-level source code representation due to compiler optimizations and hand-written assembly code. Instead of permitting control flow transfers to any function or call site, the enforced fine-grained policy is based on confining indirect edges to paths that may lead only to functions of the same prototype. Only functions with a prototype that matches the indirect call pointer prototype are considered valid targets, and this is applied to both forward and backward edges.

To reduce the over-approximation of the permitted control flow transfers even further, kCFI introduces *Call Graph Detaching*, a novel technique that detaches the direct call graph from the indirect call graph in the protected code. This prevents the issue of transitively extending the set of return targets of direct function invocations to the sets of valid return targets of indirectly invoked functions. Call graph detaching thus makes the enforced CFG more restrictive and less prone to control flow bending [13] and similar CFI bypass attacks.

Performance overhead is a crucial factor that affects the practical applicability of exploit mitigation techniques, especially for the kernel setting. To minimize the runtime overhead introduced by the extra control flow checks, kCFI leverages architectural traits, such as cache locality and no-operation instructions, to achieve better performance. This is demonstrated by the results of our experimental evaluation. When applying our prototype implementation on the Linux kernel, kCFI incurs an average overhead of 8% and 2% on micro and macro benchmarks, respectively. From a performance perspective, this makes kCFI the most efficient kernel-level CFI protection.

The main contributions of our work are:

- We present kCFI, a kernel-level fine-grained CFI mechanism that fully supports dynamically loadable modules and hand-written assembly code. kCFI does not depend on incomplete pointer analysis, nor restricts language features, as done in previous works. kCFI is orthogonal to other widely deployed exploit mitigations.
- We present a novel technique called *Call Graph Detaching*, which enhances the offered protection by enabling the construction of more precise CFGs and, consequently, enforcing a more restrictive CFI policy, with negligible additional performance cost.
- We leverage features of the x86-64 architecture to optimize the performance of kCFI while protecting the Linux Kernel.
- We have experimentally evaluated kCFI in terms of performance and security using standard benchmarks and state-of-the-art attack techniques. Our results demonstrate that kCFI offers effective protection while incurring a low overhead comparing to previous proposals, in the order of 8% and 2% for micro and macro benchmarks, respectively.

2 Background

2.1 Kernel Exploitation

Virtual memory is a well-established OS abstraction for isolating and confining user programs. Modern OSes, such as Linux [76] and Linux-based spin-offs (*e.g.*, Android [41], Chromium OS [42], Tizen [60]), Microsoft Windows [66], and the BSDs [102–104], opt for a virtual memory layout that splits the address space in two parts: (a) *kernel*

space; and (b) *user* space. The former is assigned to kernel code and data, kernel modules, and dynamic kernel memory, while the latter to user processes (*i.e.*, program code and data, heap and stack memory, shared libraries).

The separation between the two parts of the address space (*i.e.*, user vs. kernel) is enforced by two hardware features: (i) a memory management unit (MMU); and (ii) a set of CPU modes (or protection rings) [47, 90]. Typically, the OS kernel executes in the most privileged CPU mode, whereas user processes in the least privileged one—*e.g.*, the x86/x86-64 architecture provides four rings¹, with the kernel running in ring 0 and user programs executing in ring 3. The MMU, which is programmed using (privileged) special-purpose instructions, implements *hierarchical* memory protection and ensures that memory assigned to a particular ring is not accessible by code executing in less privileged rings. The end result of the above is the strong isolation of kernel space from user programs (*i.e.*, code running in user mode cannot directly access the kernel part of the shared address space).

Alas, the same property does not hold true for user space. The user space part of the address space is *weakly* isolated from kernel code. When servicing a system call, or handling an exception, the kernel is running within the *context* of a preempted process;² flushing the TLB is not necessary [69], while the kernel can access user space *directly* to read user data or write the result of a system call. Such a design facilitates fast user-kernel interactions, as well as the low-latency crossing of different protection domains.

However, the shared address space enables local adversaries (*i.e.*, attackers with the ability to run user programs) to control, both in terms of permissions and contents, part of the memory accessible by the kernel—*i.e.*, the user space part [50, 51, 99]. Hence, an attacker may execute arbitrary code, with kernel rights, by merely hijacking a (privileged) kernel control path and redirecting it to user space—thereby bypassing standard defenses like KASLR [30] and W[^]X [56, 58, 106]. Lately, attacks of this kind, known as *return-to-user* (*ret2usr*), have become the preferred way to exploit kernel vulnerabilities in modern OSes [9, 31, 48, 83, 110].

The core idea of a *ret2usr* attack is to overwrite kernel data with user-space addresses (*e.g.*, by exploiting memory corruption vulnerabilities in kernel code [83]). *Control data*, like function pointers [98], dispatch tables [33], and return addresses [93], are prime targets as they promptly facilitate code execution. Nonetheless, pointers to *essential data structures*, residing in the kernel data section or heap (*i.e.*, non-control data [108]) are also preferred targets, because they enable attackers to tamper with certain objects by mapping counterfeit copies in user space [35]. The forged data structures typically contain data that affect the control flow of the kernel, like code pointers, in order to steer execution to arbitrary points. In a nutshell, the result of all *ret2usr* attacks is that the control (or data) flow of the kernel is hijacked and redirected to user space code (or data) [51].

2.2 Control-flow Integrity

The principled approach to defend against all control-flow hijacking attacks is CFI. In foundational CFI [8], program code (indirect branches) is instrumented with checks, which ensure that dynamic control transfers are in accordance with a statically computed CFG. Yet, the effectiveness of CFI relies on the *precision* of the enforced CFG.

Weak CFI schemes, such as ROPecker [16], CCFIR [115], bin-CFI [116], kBouncer [77], FPGate [114], CFL [10], and MoCFI [27], enforce a *coarse-grained* (overapproximated) CFG due to performance [115] or deployability requirements (*e.g.*, unavailability of source code [27] or debug/symbol/relocation information [116], transparent monitoring [77]). Unfortunately, recent studies have (repeatedly) demonstrated that coarse-grained CFI is still vulnerable to ingenious code-reuse attacks [14, 28, 39, 40, 92].

As expected, this has fostered research on stronger CFI schemes; IFCC [105], {M, Pi}CFI [73, 74], CFR [84], and Lockdown [82], are all examples of *fine-grained* CFI frameworks that rely on static program analyses to generate and enforce (to the extent possible) precise CFGs. However, the construction of an accurate CFG requires solving an undecidable problem: *sound and complete pointer analysis* [88]. To make matters worse, even “ideal” CFGs include edges that are *impossible* for certain program inputs, as static analysis (be it coarse-grained or fine-grained) considers all inputs. These two fundamental limitations were systematically exploited by Carlini *et al.* [13] and Evans *et al.* [32] to bypass more strict CFI schemes. Schuster *et al.* [91] also developed techniques (albeit more esoteric) that overcome fine-grained CFI.

In the kernel setting, KCoFI [24] provides FreeBSD with support for coarse-grained CFI, whereas the system recently presented by Ge *et al.* [38] further rectifies the enforcement approach of HyperSafe [109] to implement a fine-grained CFI scheme for the kernels of MINIX and FreeBSD. PaX RAP [101] brings the fine-grained strategy to the Linux kernel by combining return address encryption with strict prototype matching to achieve CFG enforcement.

¹Modern Intel CPUs support additional modes; *e.g.*, ring -1 (hardware-assisted virtualization) and -2 (system management mode).

²In x86, the Linux kernel is placed to the upper 1GB part of the virtual address space (“3G/1G” split [19]), whereas in x86-64, it is mapped to the upper *canonical half* (*i.e.*, [0xFFFFF80000000000 : 2⁶⁴ - 1] [53]). Similar splits are used by the BSDs [64] and Microsoft Windows [67, 68].

Even so, the former has proven vulnerable to code-reuse attacks [28,39], whereas the latter (in principle) is affected by the “Control-Flow Bending” [13] and “Control Jujutsu” [32] techniques.

3 Threat Model

Adversarial Capabilities. We assume attackers with the ability to execute (or control the execution of) user programs on the OS, seeking to elevate privilege by (ab)using memory corruption vulnerabilities in kernel code [5,6]. Our model allows overwriting kernel code pointers (function pointers, return addresses, dispatch tables) with *arbitrary* values [34,98], typically through the interaction with the OS via buggy interfaces—*e.g.*, generic pseudo-file systems (`procfs` [52], `debugfs` [20]), virtual device files (`devfs` [54]), the system call layer. Code pointers may be hijacked directly [34] or controlled indirectly (*e.g.*, by first corrupting a pointer to a data structure that contains control data and subsequently tampering with its contents [35], similarly to `vtable` pointer hijacking [37,43,46,85,105,113]). Lastly, attackers can control *any* number of code pointers and trigger the kernel to dereference them on demand; note that *memory disclosure* bugs [3,4] are extraneous to our proposed scheme(s). Our adversarial model is realistic and consistent with prior work in the field [24,38].

Hardening Assumptions. We assume an OS that fully implements the W^X policy [56,58,106] preventing *direct* code injection in kernel space. In addition, we surmise an OS kernel hardened against `ret2usr` attacks; in modern platforms, we presume the existence of SMEP (Intel CPUs) [111], while for legacy systems we assume protection by `kGuard` [51] or `KERNEXEC` (PaX) [81,97]. Note, that the kernel may also support `KASLR` [30], *stack-smashing* protection [107], pointer (symbol) hiding [89], `SMAP/PAN/UDEREF` [21,70,79,80], or any other hardening feature. `kCFI` does not require or preclude such features, as they are all *orthogonal* to the proposed scheme(s) and can only increase the security of the kernel.

4 Design

Under our threat model the control flow of the kernel can be freely hijacked: *any* code pointer can (potentially) be *controlled* by the attacker. Our hardening assumptions, though, guarantee that kernel execution can no longer be redirected to code injected in kernel space or hosted in user space— W^X hinders direct code-injection in kernel space³, whereas the deployed `ret2usr` protection(s) will prevent any attempt to execute user code in kernel mode. Hence, we anticipate that attackers will be composing their shellcode by stitching together gadgets from the executable (`.text`) sections of the kernel in a ROP/JOP fashion [15,44,94], or utilize state-of-the-art code reuse techniques [14,28,29,39].

The main concept behind CFI consists in computing the control flow graph of a given program and confining all indirect control transfers to its edges. While CFI policy enforcement can effectively prevent control-flow hijacking attacks, its employment on kernel code demands specific challenges to be addressed. First, performance overheads must be minimal, preventing the use of intermediary layers between the kernel and the hardware, and requiring instead policy enforcement through lightweight approaches such as code instrumentation. Second, the enforcement must be compatible with kernel intricacies such as self-modifying code and loadable kernel modules (LKMs). Third, control transfers caused by events such as interrupts or exceptions may remain valid, even though their occurrence is not predictable in the call graph.

The scheme adopted by `kCFI` was designed to be compliant with the above challenges. By employing tag-based assertions, it supports self-modifying code and LKMs, as long as these portions of code are compiled in a compatible way. `kCFI` is fully enabled through compiler instrumentation, not requiring any supervisor module, virtualization support, or dynamic translation techniques. Although inspired by the original CFI proposal by Abadi *et al.* [8], `kCFI` differs from it as its instrumentation primitives were designed to take advantage of x86-64 architectural traits, generating close to zero memory contention. In fact, without harming any feature on the original system, `kCFI` enforces CFI with average overheads of 7%, while similar systems [24] incur costs that exceed 100%, or are only comparable after optimized with code transformations that break the above-mentioned requirements [38].

While `kCFI` enforces its policy on control flow transfers that reside in kernel address space, it relies on other well-known and widely adopted solutions to isolate memory address spaces [51,81,97,111]. This complementary approach ensures that all control transfers from kernel to user space happen through clearly defined exit points that will drop system execution privileges. Besides simplifying the protection without leaving open windows for

³We recently uncovered that many OS kernels do not properly enforce the W^X policy in kernel space [49]; major OS vendors have since taken steps to eradicate the problem [56].

(a) Return site(s) instrumentation (tag).

```

1 ...
2 callq 0xffffffff810001eb <func>
3 nopl 0x138395f
4 ...

```

(b) Epilogue(s) instrumentation (guard).

```

1 ...
2 mov    (%rsp),%rdx
3 cmpl  $0x138395f,0x4(%rdx)
4 je     <8>
5 push  %rdx
6 callq <kcfi_vhndl>
7 pop   %rdx
8 retq

```

Figure 1: Example of a kCFI return guard and tag pair.

ret2usr attacks, the scheme is compatible with all kernel control transfers, including those that are unpredictable, like interrupt handlers.

In a coarse-grained CFI system, every branch target is valid for all branches, without any distinction. Consequently, any branch is allowed to target any instruction which follows a `call` instruction, or that starts a new function. Coarse-grained CFI is no longer considered a strong defense as previous works have shown that it can be bypassed [14, 28, 39].

Fine-grained CFI schemes offer stronger protection, as they reduce the number of valid targets for each branch by applying rules to build these sets in a more restrictive way. kCFI implements fine-grained CFI by ensuring that all returns target instructions following a `call` to the returning function while indirect calls target functions meant to be reached through that invocation. As complete and sound pointer analysis is impossible [88], kCFI over-approximates the call graph by considering valid targets for an indirect call all those functions that have a matching prototype with the pointer used in the indirect `call`. In the code base we used for experimentation, the most common function prototype was `void()`. When enforcing CFI on indirect calls whose pointers had this prototype, the approach reduced the set of valid targets to approximately 0.3% of the set that would have been allowed by a coarse-grained CFI policy.

4.1 Code Instrumentation

kCFI protects the OS kernel from code reuse attacks, by ensuring that computed control transfers adhere to the CFG of the kernel, using a label-based control-flow enforcement approach [8, 24]. To this end, kCFI instruments *indirect* branch instructions (*e.g.*, `callq` and `retq` in x86-64) with control-flow assertions, in a manner similar to kGuard [51]; such control-flow assertions verify (at runtime) the target of the respective branch instructions, and authorize the control transfer(s) only if prescribed by the CFG. We refer to the code sequences used for implementing the control-flow assertions as *guards*, and to the (inlined) code labels that are checked by the guards, to validate branch targets, as *tags*.

Figure 1 illustrates a return guard and tag pair for the x86-64 architecture. In Figure 1(a), the routine `func` is invoked by a direct `callq` instruction (ln. 2), which is followed by a *return tag*, implemented as a `nopl` instruction that encodes `func`'s return ID (`0x138395f`; ln. 3). Representing the tag as a NOP instruction is important, as it *transparently* marks the return site(s) of `func` (*i.e.*, without affecting the semantics of the code). Figure 1(b) shows the corresponding *return guard* that confines a `retq` instruction of `func`. This snippet loads the intended return address from the stack into the `%rdx` register (ln. 2), dereferences it, and compares the result with the expected ID (ln. 3); the 4-byte offset in the dereference skips the `nopl` opcode, as only the encoded value (`0x138395f`) must be compared. If the two IDs match, the control jumps to the `retq` instruction and the branch is taken (ln. 4 and 8); else, the phony branch address is pushed onto the stack, and a violation handler (`kcfi_vhndl`) is invoked (ln. 5–7).

(a) Prologue(s) instrumentation (tag).

```

1 ...
2 <func>:
3 nopl    0xbcbee9
4 ...

```

(b) Indirect call site(s) instrumentation (guard).

```

1 ...
2 cmpl    $0xbcbee9,0x4(%rax)
3 je      <7>
4 push    %rax
5 callq   <kcfi_vhndl>
6 pop     %rax
7 callq   *%rax
8 nopl    0x138395f
9 ...

```

Figure 2: Example of a kCFI entry-point guard and tag pair.

Along the same vein, Figure 2 depicts an entry-point guard and tag pair for the x86-64 architecture. The prologues of routines that can be indirectly invoked are marked with an *entry-point tag*, similarly to return sites; Figure 2(a) shows the entry-point tag of routine `func`, also implemented as a `nopl` instruction that (transparently) encodes the routine’s entry-point ID (`0xbcbee9`). Figure 2(b) illustrates the corresponding *entry-point guard* that confines an indirect `callq` instruction to `func`. Assuming that the address of `func` is loaded in register `%rax`, this snippet dereferences `0x4(%rax)` and compares the result with the expected ID (`0xbcbee9`; line 2). Again, if the two IDs match, the control jumps to the `callq` instruction and the indirect invocation of `func` takes place (lines 3 and 7); else, the bogus branch address is pushed onto the stack and `kcfi_vhndl` (violation handler) is invoked (lines 4–6). Note that `callq *%rax` is followed by a return guard (*i.e.*, `func`’s return guard; return site, line 8).

The end result of the above (control-flow) confinement scheme is the following: (a) `retq` instructions can only transfer control to the return site(s) of the routine they belong to (*e.g.*, the `retq` instruction of `func` can only transfer control to the return sites of `func`; Figure 1); and (b) indirect `callq` instructions, which correspond to function pointers, are paired with the beginning of the routines that can be indeed invoked through the respective function pointer (*e.g.*, the `callq` instruction of Figure 2(b) can only transfer control to the prologue of `func`).

kCFI’s guards are designed so that every confined branch instruction is paired with a call to the violation handler. By using this approach, instead of having a single (global) call to `kcfi_vhndl` (to which every guard transfers control upon an assertion failure), it is possible to precisely trace the violations by combining the pushed parameter and the violation handler’s own return address. We found that this configuration increases the overall overhead by $\sim 2\%$, making its benefits more appealing than its costs.

Performance Requirements. The proposed tag-based scheme employed by kCFI conforms to the hard performance requirements imposed by OS kernels. Specifically, in the x86-64 architecture, kCFI is significantly more efficient than previous approaches, as demonstrated by the results presented in Section 6. Given the multi-level and inclusive nature of the cache hierarchy of Intel CPUs, the proposed guards do not generate compulsory cache misses; if a cache miss happens while dereferencing the branch target for validation, the only consequence is the anticipation of a load that would otherwise occur while branching.

kCFI also implements tags in a more efficient way than previous tag-based schemes. The original CFI proposal by Abadi *et al.* [8] uses `prefetch` instructions to mark valid branch targets (*e.g.*, encode return IDs). Although such instructions do not change the semantics of a program, they do leave a footprint on cache organization and affect memory contention [112]. By employing `nopl` instructions, kCFI is capable of marking valid targets with close to zero side-effects (see Section 6). To the best of our knowledge, kCFI is the most efficient tag-based CFI scheme for the kernel setting.

Compatibility Requirements. Many of the previously-proposed CFI schemes for low-level software, like OS kernels, implement CFG validation techniques that rely on converting indirect branches into jump tables based on *restricted pointer indexing* [38, 109]. The kernel-level CFI scheme proposed by Ge *et al.* [38] uses this approach, and, to achieve good performance, has to rely on code optimizations (*e.g.*, function pointer constification) that heavily depend on the intricacies of certain code bases (*i.e.*, FreeBSD and MINIX). In antithesis, kCFI is capable of protecting OS kernels despite of any optimization opportunity, while achieving better performance.

4.2 Fine-grained CFI Policies

For building its fine-grained policy, kCFI relies on a call graph used as a reference for valid branches on the protected program. To build this call graph, kCFI relies on two different types of analysis. First, source code analysis provides all high-level information regarding functions, function pointers, and prototypes. Second, a binary analysis performed on a compiled version of the program allows fitting the call graph accordingly to back-end optimizations and transformations that occur during linking. Building the call graph through combining both analyses provides a precise and reliable call graph, as required for the development of a fine-grained CFI.

For every function represented on the call graph, kCFI creates an exclusive return tag. For every unique prototype respective to an indirect invocation pointer, kCFI creates one return tag and one entry-point tag. Whenever marking the code with return tags, kCFI gives precedence to those respective to the indirect invocation prototype. Entry-point tags are used to mark functions as valid targets for indirect calls. This approach guaranties a fit restrictiveness while ensuring that a function return may be allowed to all its valid return targets, irrespectively of whether it was invoked directly or indirectly. During instrumentation, kCFI considers indirectly invocable all those functions whose prototype has a corresponding function pointer prototype.

When placing tags, kCFI parses all `call` instructions inside each function. If the `call` is indirect, then the return tag respective to the indirect invocation pointer prototype is placed right after it. If the `call` is direct, then kCFI first verifies if the function being called is also indirectly invocable. If it is, then the return tag placed is the one correspondent to the indirect invocation pointer prototype. Otherwise, then the return tag for the function is used. kCFI also checks if each parsed function is indirectly invocable. If it is, then the entry-tag that corresponds to its prototype is placed in the function’s prologue.

Next, kCFI places guards in each function. The tags checked by return guards are picked through a logic similar to the one described above: whenever a return guard for an indirectly invocable function is generated, the tag used is the one corresponding to an indirect invocation pointer’s prototype that matches the function’s prototype. If the function is not indirectly invocable, the tag used is the one respective to the function. For generating entry-point guards, kCFI checks the prototype of the indirect invocation pointer used in the `call` and picks the entry-tag relative to it. This scheme ensures the proper bonding of all indirect branches and their relative targets.

Although the work by Abadi *et al.* [8] suggests the use of a *shadow stack* to refine returns, it is known that such structures have high performance costs [26], which are prohibitive in the kernel context. Besides that, a shadow stack imposes limitations to control flows that diverge from a strict `call/ret` parity, which are very common in kernel code. For these reasons, we do not implement a shadow stack in kCFI.

Call Graph Detaching

If a function is both directly and indirectly invocable, its return guards will be generated with the return tag respective to its prototype, irrespectively of whether it was directly or indirectly invoked. This creates a situation where transitively all instructions after a direct call to a function become valid return points to other functions with a similar prototype. This problem stretches the CFI policy and makes it more prone to bending attacks [13].

This problem is illustrated in Figure 3. The code snippet in Figure 3(a) invokes `foo()` both directly and indirectly. The code snippet in Figure 3(b) presents the return guard for the function `foo()` which checks for the tag placed after both direct and indirect calls to `foo()`. This not only allows `foo()` to return to both call sites, irrespectively of how it was invoked, but it may also allow a *different* function with the same prototype as `foo()` to return to the call site of the direct invocation of `foo()`, in a clear violation of the valid control flow.

To mitigate this problem, kCFI follows a novel approach by cloning functions instead of merging all valid return targets. In this way, a function named `foo()` is cloned into a new function called `foo_direct()`, which has the same semantics but checks for a different tag before returning. All direct calls to `foo()` are then replaced by calls to `foo_direct()`, and the tag placed after the call site is the one that corresponds to `foo_direct()`. This approach, which we call optimization *Call Graph Detaching* (CGD), detaches the kernel’s direct call graph from the indirect call graph, preventing the need for merging and the consequent over-approximation caused by tag transitiveness. When applied to the code of Figure 3, CGD results in the code presented in Figure 4. The optimized version brings the replaced `call` instruction and the respective tag (4(a)) and both original `foo()` and cloned `foo_direct()` functions (4(b,c)).

Call graph detaching is applied only on functions that can be invoked both directly and indirectly. For that, the algorithm checks the existence of pointers with a matching prototype and direct invocations before cloning a function. As clones are used to replace the targets of direct calls, pointer operations that may end up targeting the function are not harmed by the scheme. An analysis of how CGD refines the granularity of the applied CFI policy is provided in Section 6.

(a) function *foo()* invocation

```

1 ...
2 callq 0xffffffff810001eb <foo>
3 nopl 0x1383ddd
4 ...
5 movl rcx, 0xffffffff810001eb <foo>
6 callq *rcx
7 nopl 0x1383ddd
8 ...

```

(b) return guard on *foo()*

```

1 ...
2 mov (%rsp),%rdx
3 cmpl $0x1383ddd,0x4(%rdx)
4 je <8>
5 push %rdx
6 callq <kcfi_vhndl>
7 pop %rdx
8 retq

```

Figure 3: Example of valid merged return targets.

4.3 CFI Map

To create its fine-grained CFI policy, kCFI uses a data structure called *CFI Map*, which is an augmented call graph built using source code and binary analysis. Besides function invocation relationships, this structure also holds function prototype information that enables mapping which functions may be called indirectly, and symbol information that permits correctly mapping functions hidden behind aliases.

To construct a complete CFI Map, the respective binary to be protected needs to be analyzed, and this is done through an early compilation. The construction process compiles the entire kernel while performing source code analysis. This compilation also instruments the generated code with identification marks that enable disambiguation of functions when their names collide in the final binary. When the kernel binary is ready, it is analyzed to fill any information gaps left during the source code analysis phase.

5 Implementation

kCFI was implemented in the form of a compiler infrastructure composed of a set of complementary tools that perform code analysis and CFI instrumentation. The whole set can be classified into four different groups of tools: (i) source code analysis, (ii) binary analysis and CFI Map construction, (iii) Assembly patchers, and (iv) CFI instrumentation. From these groups, (i) and (iv) were implemented as LLVM compilation passes, and each implies a full compilation of the source code, while (ii) and (iii) are a group of tools written in C++ and Lua [87]. kCFI can be understood as a pipeline in which each group of tools is a stage that follows the above order. A detailed illustration of the different stages is presented in Figure 6, and is described in detail in the following. Overall, the goal of stages (i) and (ii) is to build a CFI Map for the whole kernel code, while (iii) and (iv) use the resulting CFI Map to instrument the kernel code with tags and guards.

Assembly Language Support

One of the drawbacks of using LLVM-based instrumentation is that assembly sources are not touched, as this kind of code is directly translated into binaries without having an intermediate representation (IR) form. The kernel has a significant part of its code written in assembly, which includes many indirect branches. While applying CFI, if such code is left unprocessed, two major problems arise: (i) indirect branches in assembly sources are left unprotected, and (ii) tags are not placed, breaking compatibility with C functions returning to assembly, or with assembly functions being called indirectly from C code. kCFI tackles this problem through the automatic rewriting of the assembly sources assisted by information extracted during code and binary analysis, as explained in the following subsections.


```

(a) function foo() invocation
1  ...
2  callq 0xffffffff8100033a <foo_direct>
3  nopl  0xaff883
4  ...
5  movl  rcx, 0xffffffff810001eb <foo>
6  callq *rcx
7  nopl  0x1383ddd
8  ...

(b) return guard on foo()
1  ...
2  mov   (%rsp),%rdx
3  cmpl  $0x1383ddd,0x4(%rdx)
4  je    <8>
5  push  %rdx
6  callq <kcfi_vhndl>
7  pop   %rdx
8  retq

(c) return guard on foo_direct()
1  ...
2  mov   (%rsp),%rdx
3  cmpl  $0xaff883,0x4(%rdx)
4  je    <8>
5  push  %rdx
6  callq <kcfi_vhndl>
7  pop   %rdx
8  retq

```

Figure 4: Example of call graph detaching (CGD).

CFI Map

The CFI Map describes the kernel’s call graph using four entities: (i) Nodes, which represent single functions; (ii) Clusters, which represent sets of functions with the same prototype; (iii) Edges, which represent a branch from a node to another node or to a cluster; and (iv) Aliases, which map symbols that may have different names but are equivalent in the final binary.

- **Nodes.** These entities represent functions and hold as attributes their identifier, function name, prototype, and module. Each node also contains a tag which will be later used to validate allowed returns to the function.
- **Clusters.** These entities represent a group of functions that have the same prototype. Each cluster has its identifier and information on the prototype it represents. Clusters also hold two tags, one used to validate allowed return points for the functions in the cluster, and another to mark the entry-points of these functions as valid targets for indirect calls.
- **Edges.** Edges represent the invocation relationship between nodes and clusters. An edge has an identifier for itself and holds the identifiers for the node that corresponds to the edge’s origin and for the node or cluster of the edge’s target. Edges also have a type attribute that defines them as a direct or indirect call.
- **Aliases.** Aliases represent symbols that can hide another symbol during compilation time.

Figure 5 shows an example of a CFI Map construction. The three functions in the source code (a) are represented in the graph (b) as nodes (circles). The gray rectangle in the graph represents a cluster, which includes the nodes `i32 A(i32)` and `i32 B(i32)`. The solid edge represents the direct call to the function `void C(i32)`, and the dashed one represents the indirect call to functions inside the cluster, which has the prototype `i32 (i32)`.

By observing Figure 5 it is possible to infer important information for call graph enforcement: (i) there is a direct call from node A to C, implying a return from C to A that must be allowed; (ii) there is an indirect call from B to the cluster, so the returns of all functions in the cluster (A and B) to B must be allowed; and (iii) the indirect call from B must also be allowed to all functions in the cluster. Figure 5(c) shows the corresponding CFI Map file data structure.

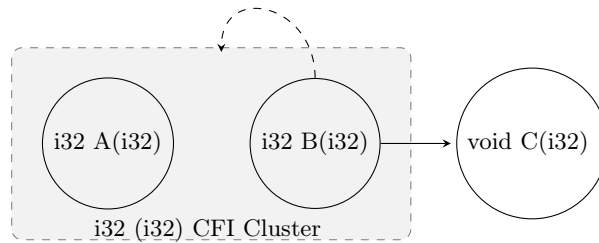
(a) Example source code.

```

1  int A(int x){
2      return x*x;
3  }
4  int B(int y){
5      int(*f)(int);
6      f = &A;
7      C(30);
8      return 7 * f(y);
9  }
10 void C(int z){
11     while(1){ };
12 }

```

(b) Resulting CFI Map.



(c) Resulting CFI Map data structure.

Nodes				
Identifier	Name	Prototype	Module	Return tag
290f2fd5	A	i32 (i32)	ex.c	1dc2aaf0
7d63f629	B	i32 (i32)	ex.c	6e28b9d1
6ba8458b	C	void (i32)	ex.c	164e44a8
Clusters				
Identifier	Prototype	Entry-point tag	Return tag	
6a8597ea	i32 (i32)	69e1b040	46068a5c	
Edges				
Identifier	From	To	Type	
7dcdc019	7d63f629	6a8597ea	indirect	
7728cc01	7d63f629	6ba8458b	direct	

Figure 5: Example of CFI Map construction.

kCFI Pipeline Overview

Figure 6 describes how the different stages of kCFI are connected. In this diagram, rectangles represent files used or generated by the pipeline stages, dashed arrow shapes represent a full compilation through LLVM, dashed circles represent an offline step, and solid arrow edges describe which files are used or generated by each step.

Initially (1), the kernel source code is compiled with LLVM, generating a *vmlinux* file and multiple CFI Maps, one for each compiled module. The compiler also performs two tasks: instruments *vmlinux* with a unique identifier on the first instruction of every C function, and stores a catalog of all function declarations seen during compilation. All CFI Maps are merged (2) into one single CFI Map, which is used together with the kernel binary to uncover all assembly functions (3). These are the ones not instrumented with a unique identifier during the previous compilation. The outcome of this stage (4) is then analyzed by a tool that retrieves all direct `call` targets and, by checking their unique identifiers, maps every direct edge. Finally, some fixes to the CFI Map are applied (6) to support certain corner cases, such as `syscall` functions, which are invoked through pointers that do not have a matching prototype.

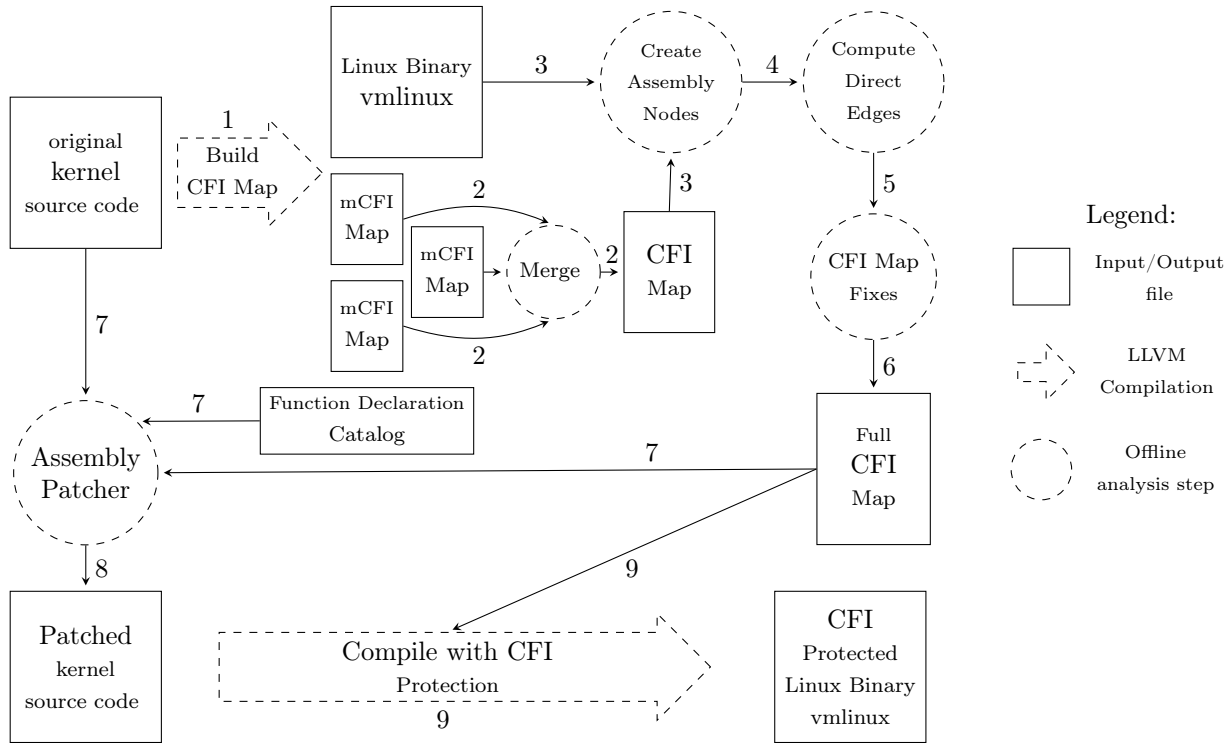


Figure 6: kCFI Pipeline

Once the CFI Map is complete, it is first used as a reference to patch the assembly files. By combining its information with the catalog of declarations (7) built during the first compilation (1), the assembly files present in the original kernel source are rewritten with proper tags and guards. The patched source (8) is then ready to be compiled by LLVM, a process that instruments C functions using information from the CFI Map and generates the final protected `vmlinux` binary.

5.1 Source Code Analysis

Source code analysis is the first stage in kCFI’s pipeline and it is implemented as a compilation passes in LLVM. It is represented in Figure 6 by the arrow shape (1). The goal of this stage is to begin the construction of the CFI Map by adding source-level information retrieved from all functions in each compiled module. The kernel binary that results from this compilation is used by the following binary analysis stage.

During this process, each compiled function is stashed in the CFI Map as a node. From each function, all indirect calls are parsed and used to create indirect edge entries. Each edge’s origin is the node that represents the function being parsed and its target is a cluster that represents the functions with the same prototype as the pointer. A new cluster is created if such a cluster does not already exist in the CFI Map. This process also stores all aliases and the respective symbols overridden by them. This is important to allow further CFI instrumentation to resolve symbols and match a branch’s target precisely, as it appears in the final binary.

Since every direct call also represents an indirect branch, i.e., the return from the callee towards the caller, it is also important to map those edges. This pipeline stage is not appropriate for performing this mapping as it may result in incomplete edges, because (i) by the time the module is analyzed, some callee functions may have not yet been compiled, and thus their nodes will not exist in the CFI Map; and (ii) as some symbols have weak linking properties, if this is the case for the callee in the analyzed edge, it cannot be correctly identified prior to linking.

To ensure that direct edges are precisely mapped during binary analysis, the compilation process in this stage also instruments the prologue of each function with its respective node identifier, encoded similarly as done with tags, as described in Section 4. This instrumentation does not concern CFI enforcement, but allows the binary analyzer to directly correlate functions in the binary with node entries in the CFI Map, independently of aliases, optimizations, and linking properties.

```

1  mov    (%rsp),%rdx
2  cmpl  $0x138395,0x4(%rdx)
3  je     9
4  cmpl  $0x11deadca,0x4(%rdx)
5  je     9
6  push  %rdx
7  callq <kcfi_vhndl>
8  pop   %rdx
9  retq

```

Figure 7: An example of a guard with secondary tag support.

By the end of the source code analysis, the data structure being built already holds node entries for all functions written in C, edge entries for all indirect calls present in C functions, cluster entries for all prototypes used to declare function pointers, and a map of all aliases with their respective masked symbols. This stage also generates a fully compiled kernel binary in which every C function is instrumented with a unique identifier.

5.2 Binary Analysis and CFI Map Fixes

The binary analysis phase complements the constructed CFI Map with information extracted from the compiled kernel. This stage comprises three main steps, which are represented in Figure 6 by the edges 4, 5 and 6.

The first step performed during binary analysis is a search for functions which were not instrumented with an identifier in the previous stage. A function left not instrumented means that it was not touched by LLVM in kCFI’s first stage, because it was originally written in assembly, and not C. Node entries for all these functions are added to the CFI Map, as part of the assembly nodes list. The address of the first instruction in each assembly function is also kept as the function’s identifier.

The second step involves parsing all direct call instructions in the binary. By following the instruction’s target address, it is possible to reach the call’s destination, retrieve its identifier, and create a direct edge entry in the CFI Map with both the origin and target fields filled. If the target does not have an identifier, kCFI assumes that it is an assembly function and retrieves the identifier from the list of assembly nodes using the address of its first instruction.

CFI Map Fixes

The third step of the binary analysis stage applies fixes to the CFI Map, making it compliant with various kernel corner cases.

Alternative Calls. The Linux kernel dynamically replaces the targets of certain direct `call` instructions with more efficient implementations of the respective functions, depending on the presence of specific CPU features. The whole set of functions that may potentially be a target for a given `call` must thus be allowed to return to the same points, similarly to how it is performed for clusters; consequently, their tags must be merged. This is achieved by setting a unique tag value in the node entries that represent these functions in the CFI Map.

Sysrem Calls. The functions that integrate the set of `syscalls` have different prototypes. Besides that, `syscalls` can be invoked through both regular direct calls or indirectly, through the *syscall dispatcher*. As only one tag can be placed after the `syscall` dispatcher, fixing its verification requires merging all clusters that hold a prototype that matches a `syscall`’s prototype. This solution causes a broad loosening of the call graph, as it results in a broad merging of different clusters, and also creates clusters for functions which are never indirectly called from places other than the `syscall` dispatcher.

Instead of merging these clusters, kCFI handles `syscalls` in a special way. First, the *syscall table* file is parsed, and a catalog of `syscalls` is built. A generic tag for the `syscall` dispatcher is also created. Second, while compiling the kernel’s source code, `syscalls` are compiled with a different kind of guard that allows returning upon validation of one among two different tags. The first tag that is checked is the one that corresponds to the function (either its node or cluster return tag, and the second is the generic tag created for the `syscall` dispatcher. An example of such a “specialized guard” with secondary tag support is shown in Figure 7.

By the end of this stage, the CFI Map holds node entries for all functions in the binary, with prototype information for those written in C; cluster entries for all prototypes used to declare function pointers; edge entries for all the indirect calls in C code; and edge entries for all direct calls present in the final binary, with no ambiguity due to weak linking or aliasing.

5.3 Assembly Code Patching

Once the CFI Map is complete, an *Assembly Patcher* rewrites the assembly files that exist in the kernel source tree. Before changing the code, the tool needs to retrieve the tags that will be used during code instrumentation in three different circumstances.

Direct Calls from Assembly Code. Assembly code may call functions that check for a tag before returning. Consequently, calls from assembly code must also be followed by a tag respective to the called function. The Assembly Patcher collects these tags by parsing the target of every `call` instruction in the source code and retrieving its node from the CFI Map.

Indirect Calls to Assembly Functions. Assembly functions may be called indirectly. Consequently, they need to have a tag in their prologue to allow them as indirect call targets. As no prototype information is available in assembly code, retrieving this tag requires parsing all function names from the source code and searching for them in the declarations catalog created during the source code analysis to identify their prototypes. If a matching declaration with a respective cluster is found, then its cluster entry from the CFI Map is used. The system has been designed to prompt the user in case the search returns more than one match, but such cases were not seen during our evaluation.

Return Instructions in Assembly Code. Assembly functions also must be protected against control-flow hijacking; it is thus important to instrument their indirect branches with guards. For these cases, all `ret` instructions are parsed and the name of the function to which they belong is searched in the declarations catalog. If a match with a respective cluster exists, the cluster is retrieved from CFI Map. Otherwise, the function's node is retrieved.

After collecting all the tags, the Assembly Patcher is capable of rewriting the source files by correctly placing the tags after every direct `call`. Placing tags in the prologue of assembly functions is achieved by replacing the macro `ENTRY` with a specially crafted macro `ENTRYcfi`. While the first is regularly used in the Linux kernel source code to mark the beginning of functions and appropriately create their symbols, the later was crafted to extend `ENTRY` in taking one extra argument to be placed as a tag when the macro is expanded. This way, for placing the tags in prologues, the Assembly Patcher rewrites the code replacing `ENTRY` for `ENTRYcfi` macros with the corresponding tag, whenever it is required. Assembly return guards are placed in a similar way. All functions are rewritten with the instructions of the corresponding guard preceding their `ret` instructions.

Some assembly functions are generated through the expansion of macros. In these cases, there exist `call` instructions whose targets are passed as a macro parameter, making it impossible to add the tag directly in the macro source, as previously done. For such cases, kCFI has a crafted macro which is a variation of the original, but with one extra parameter corresponding to the tag to be placed. The process of rewriting consists of parsing the original macro invocation, retrieving the symbol which will be expanded as a `call` target, searching the CFI Map for its corresponding tag, and then replacing the original macro in the source code with the crafted macro, also adding the tag among its parameters. Placing guards on macro generated functions again follows the same logic.

Assembly inlined into C functions is not general enough to be tackled in an entirely automated way, and is handled by directly patching the source code. For these cases, we prepared a set of patches that already have placed tags and guards accordingly, but with a generic value. Before applying the patches, kCFI searches the CFI Map for the particular tags of each case and uses them to replace the general value. Finally, the file that holds the *syscall dispatcher* is also rewritten with its tag correctly placed.

As assembly code does not provide information about pointers' function prototypes, the methods proposed so far are insufficient to automate the generation of indirect call guards in such files. For the code base we used during our tests, kCFI protected 139 indirect branches in assembly modules. kCFI missed 6 instructions which involve no hazard, being part of initialization routines only invoked during boot time, 5 indirect calls that are inherently protected by being implemented in the form of verified target tables, and 5 indirect calls whose pointer prototypes cannot be inferred statically, but were feasible to be patched and moved to read-only data sections. Instructions belonging to code which is compiled along with the kernel, but which are not linked into the final binary, such as user-space vDSO, were intentionally skipped as they cannot be hazardous.

5.4 CFI Instrumentation

The last stage of the kCFI pipeline compiles the kernel code with CFI protection. As assembly code was protected in the previous stage, C code is now instrumented with tags and guards through an LLVM back-end pass. For this process, kCFI relies on the CFI Map built during the previous steps. As no information is written to the CFI Map, this stage can be run concurrently with no harm, allowing parallel compilation. Applying or not the CGD optimization is a configuration option through the use of a special compilation flag.

To enforce CFI, kCFI parses all instructions in the code being compiled while they are still in LLVM Machine IR form. This abstraction is much closer to the final binary than high-level languages, allowing instrumentation to interpose instructions more precisely. Also, still being an LLVM representation, this allows the use of the compiler’s API to retrieve high-level information, such as prototypes of pointers used in indirect calls, which are crucial for CFI enforcement.

The default violation handler function, which is invoked when a CFI assertion fails, was implemented to display meaningful debug information and then halt the system. Different violation handlers can be implemented in custom forms, e.g., for debugging purposes or even to allow fail-safe routines. These functions are written in C, as a regular kernel module, and are easily replaceable.

If CGD is in use, an extra compilation pass is needed before the CFI instrumentation. This pass checks all functions to identify the ones that are callable both directly and indirectly. This is done by checking the CFI Map for the existence of both a cluster with the matching prototype and at least one direct edge towards the function. If a function meets both requirements, it is cloned into a new function that is set as never indirectly invocable. While defining cloning targets, functions that are declared but not implemented are also considered, ensuring CGD applicability throughout all modules. Before proceeding, calls whose targets were cloned are replaced by calls to the cloned function, ensuring that one function is only callable either directly or indirectly, never both ways. The generated code is then delivered for CFI instrumentation, and this will place tags considering the branch’s target.

During instrumentation, indirect branches are preceded with guards and indirect branch targets are marked as valid with tags, as described in Section 4. All values used to generate tags and guards are retrieved from the CFI Map. Whenever consulting it, kCFI verifies the alias entries to avoid ambiguities that could lead to wrong instrumentation. No link-time optimization or binary modification is required after the kCFI pass processes the LLVM IR.

6 Evaluation

kCFI was evaluated across three different perspectives: performance impact, security protection, and size overhead. To that end, we used as code base the Linux kernel version 3.19.0, with the LLVMLinux [59] patches applied. The whole kernel was compiled with a large number of built-in drivers and functionality, resulting in a realistic code base for evaluation purposes. Besides the original files, a CFI-specific module that holds the violation handler function was also included in the source set and was linked in the final binary. All tests were performed on a system equipped with a quad-core 3.40GHz Intel(R) Core(TM) i7-6700 processor, 32GB of RAM, and a 500GB SSD hard drive. The Debian GNU/Linux 8 (Jessie) was running on top of the tested kernel.

To verify the performance degradation cause by kCFI’s instrumentation, we tested three different versions of the Linux kernel: (i) *Pure*, the original kernel compiled with LLVM; (ii) *kCFI*, the kernel compiled with CFI protection; and (iii) *kCFI+CGD*, the kernel compiled with CFI protection using the call graph detaching optimization.

To assess the security benefits introduced by kCFI, we use the *average indirect target reduction* (AIR) metric [116], which measures the program’s restrictiveness according to the applied CFI policy in terms of call graph pruning. We also use the *average indirect targets allowed* (AIA) metric [38], which captures the number of permissible targets for every indirect target.

6.1 Performance Evaluation

Micro-benchmarks

The micro-benchmark LMBench [65] was used to verify kCFI’s impact on kernel operations. From the whole benchmark, a subset of applications focusing on OS capabilities was selected, allowing the measurement of latencies through the execution of null syscall; I/O critical syscalls (read/write, fstat and select); open/close syscalls; signal handler installation, process creation followed by exit, execve and /bin/sh; context switching between processes; select syscall on 100 file descriptors; and page fault handling and inter-process communication with socket and pipe. Communication throughputs through pipes, unix sockets (AF_UNIX), and TCP sockets were also measured.

Figure 8 shows the performance overhead of kCFI and kCFI+CGD over Pure. The micro-benchmark tests are classified in latency and communication throughput overhead. For latency, kCFI incurs an average overhead of 8%, while kCFI+CGD 7%. The maximum overhead of both configurations reaches 33%. For communication throughput, kCFI incurs an average overhead of 2%, while kCFI+CGD did not exhibit any discernible overhead.

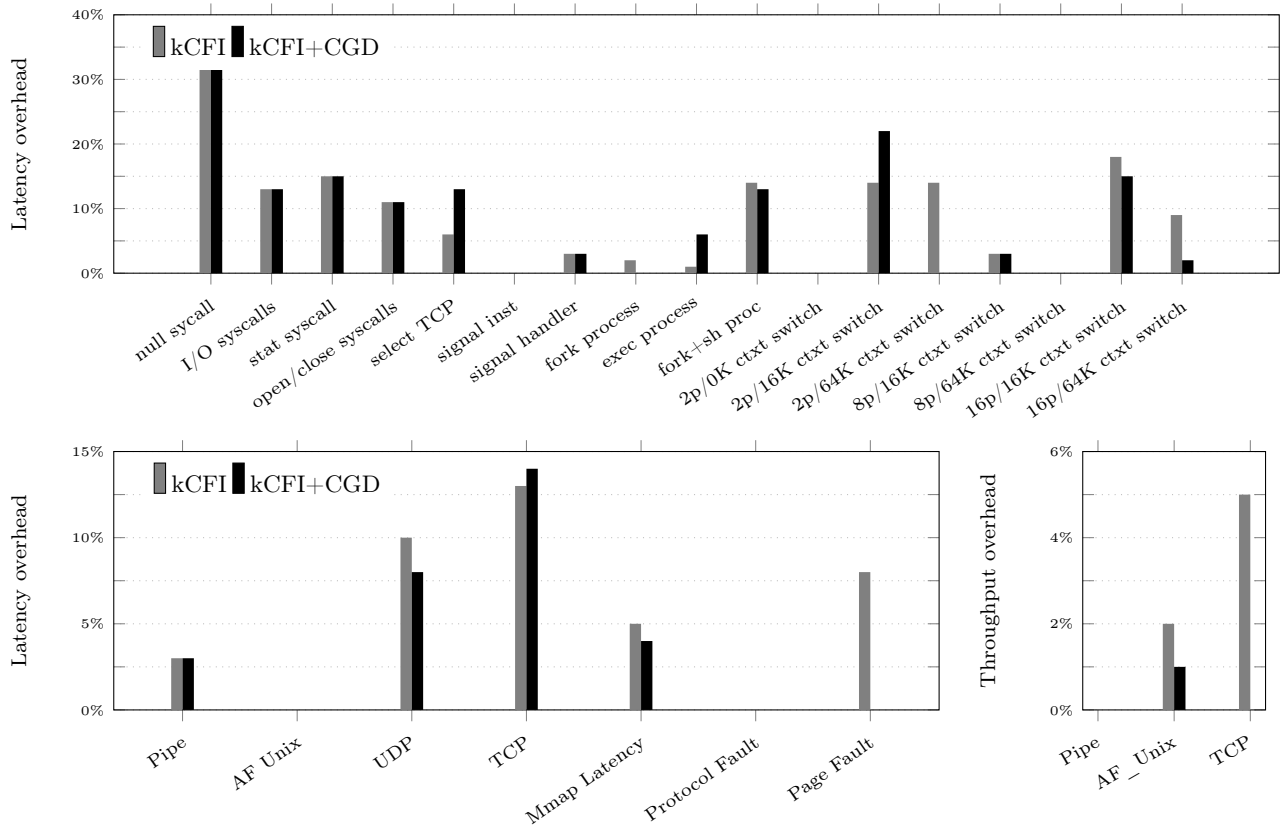


Figure 8: Performance overhead of kCFI on LMBench.

Macro-benchmarks

To measure the effects of kCFI on user-space applications running on top of an instrumented kernel, we used tests from the *Phoronix Test Suite* [55]. The set was composed by the benchmarks *IOZone*, running 1MB, 4Kb and 64Kb read/write operations on a 512MB File, *Linux Kernel Unpacking*, *PostMark*, *Timed Linux Kernel Compilation*, *GnuPG* encrypting a 1GB file, *OpenSSL* running 4096-bit RSA, *PyBench*, *Apache Benchmark*, *PHPBench*, *Dbench* and *PostgreSQL* running read-only/read-write operations under heavy contention on disk, cached, and in buffer.

Figure 9 shows the overhead for each test. The average overhead observed for the whole test suite was 2% for kCFI and 3% for kCFI+CGD, with a maximum of 20% and 19%, respectively, both for the Apache Benchmark. Besides Apache, all other applications exhibited overheads below 10%. To assess the reasons for the outlier, we also run the test *NGINX*, which is another web server application. This test exhibited an average overhead of 20%, confirming that the higher observed values are due to their frequent interactions with kernel capabilities, turning them into applications more sensitive to such instrumentation.

6.2 Security Evaluation

The threat model considered assumes that both W^X and $ret2usr$ hardening features are in use, leaving an attacker with no opportunities for code injection neither in kernel or user space. Successful exploits are then limited to code-reuse strategies over kernel instructions, commonly executed through return-oriented programming-based attacks [86]. These attacks are built upon chaining small instruction sequences terminated with an indirect branch. These instruction sequences are called gadgets, and each performs a small operation. Through chaining many gadgets, attackers achieve Turing-complete computation [12].

The goal of kCFI is to protect the system against control-flow hijacking by preventing code-reuse attacks through limiting valid targets on each indirect control flow transfer. Such enforcement turns the attack unfeasible as the gadgets become unreachable through corrupted indirect transfers, severely limiting possibilities of recombination to achieve the desired computation.

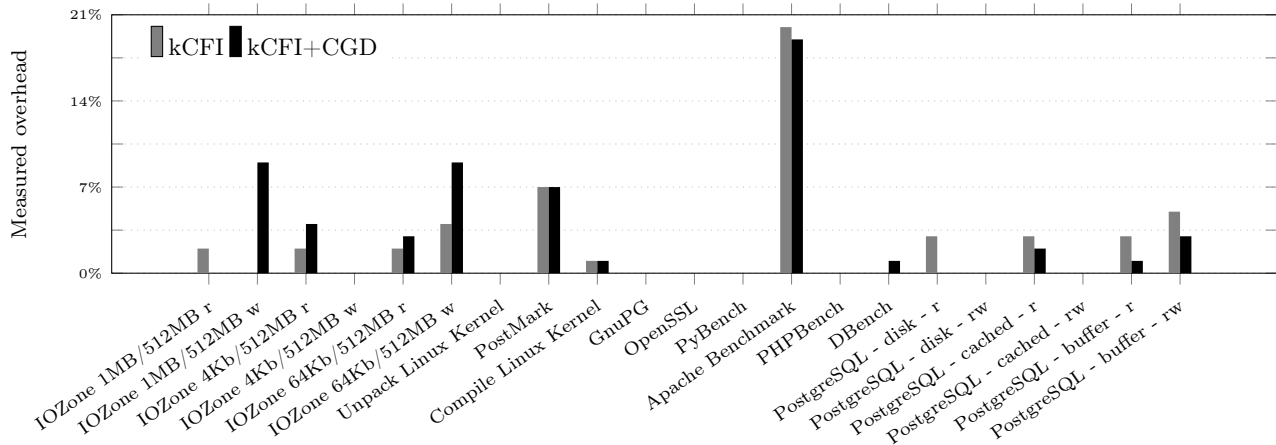


Figure 9: Performance overhead of kCFI on Phoronix.

Return-oriented Programming Attacks. Gadgets used during return-oriented programming attacks are small instruction sequences terminated with a `ret` instruction, which will redirect the execution towards the next gadget. As kCFI only allows indirect branches to matching tags, most of these gadgets become unreachable and unusable. Gadgets that remain useful for fortuitously being preceded by a tag cannot freely chain others, as its closing `ret` will only be allowed to redirect control to a reduced set of valid targets.

Unintended Gadgets. As the x86 architecture does not require executed instructions to be aligned, it is possible to retarget an indirect branch to unaligned addresses that may contain unintended gadgets across original kernel instructions [94]. As kCFI enforces all indirect transfers to target an aligned tag, it requires unintended gadgets to also be preceded with unintended tags, restriction that minifies available gadgets to the point of preventing Turing-complete computation through this technique.

Coarse-grained CFI Attacks. Coarse-grained CFI restricts control flow based on loose approximations of the applications CFG [24, 27, 115, 116]. It has been shown that these policies are yet permissive, and systems protected through such weak CFI schemes remain exploitable through the use of special gadgets which are compliant with the restrictions imposed [14, 28, 39]. kCFI is not vulnerable to these attacks as it applies a more restrictive CFG while building its policies, reducing available gadgets and precluding the remaining from being freely chained.

Evasion Gadgets. Some CFI systems are based on branch monitoring, detecting attacks by matching ROP-characteristic execution patterns [16, 77]. Such protections were shown to be flawed, as attackers can use evasion gadgets that will diverge the exploit’s control-flow path from what is considered an anomaly, turning the attack indistinguishable [14, 28, 39, 40, 92]. Such attacks are indifferent to kCFI as it detects attacks through stateless statical instrumentation instead of dynamic observation of execution patterns.

Call-oriented Programming Attacks. Call-oriented programming-based attacks employ gadgets which are terminated with an indirect `call` instruction instead of a `ret`, corrupting its function pointers to chain other gadgets consecutively [14]. As kCFI requires all indirect calls to target the first instruction in a limited set of functions, it diminishes the number of available gadgets useful for these attacks turning them insufficient for achieving Turing-complete computation.

Jump-oriented Programming Attacks. Jump-oriented programming-based attacks use a particular gadget called *dispatcher gadget*, that employs an indirect jump to redirect control flow through a maliciously crafted list of functional gadgets, chaining them to achieve Turing-complete computation [11]. Although kCFI does not instrument jump instructions, we ensured that all indirect jumps present in the kernel are unusable for Jump-oriented programming purposes. The largest part of these instructions use verified index registers only allowed to target addresses confined to an unwritable memory region. A minor portion has its target set through a hard-coded immediate just a few instructions before being used. Both situations leave no opportunity for indirect jump target corruption.

Control-Flow Bending and Control Jujutsu. Exploits that employ code-reuse attacks whose flows are confined to the valid CFG paths were shown to be deployable against systems protected with user-space fine-grained CFI. Both Control-Flow Bending [13] and Control Jujutsu [32] are specializations of non-control-data attack that corrupts the arguments of a specific function for performing malicious computation and then manages to divert control flow through a valid path to invoke it. kCFI does not fully close the matter against these attacks, but it raises the bar

by enforcing more restrictive CFGs through applying the CGD optimization. This optimization creates obstacles for the attacks, especially against control flow bending, as it depends on corrupting returns to construct loops.

Although possible in theory, a better understanding of how these techniques affect kernel code is required prior to assuming their efficiency in this context. Control-Flow Bending, for example, is demonstrated through (but not limited to) the employment of a technique called *printf-oriented programming*, that exploits a specific format string character to create memory write operations. As such character and its respective functionality are not available in `printk`, the kernel version of `printf`, the attack needs to be deployed through different and less common functions. Similarly, Control Jujutsu depends on the possibility of indirectly invoking functions whose parameters can be controlled to cause arbitrary computation. Both invocation possibility and function existence are uncertain in the kernel context and require a deeper analysis.

Real-world Exploits. To assess the effectiveness of kCFI against real-world attacks, we used the ROP exploits for CVE-2010-3301 [17] and CVE-2010-3904 [18] by Kemerlis *et al.* [49], targeting Linux v2.6.33.6, as well as their custom exploit for v3.12. We first verified that the exploits were successful on the appropriate kernels, and then tested them on the same kernels armed with kCFI. In all cases, the respective exploits failed, as the ROP payloads relied on pre-computed gadget addresses, none of which remained a valid (control-flow) target under kCFI.

CFI Metrics

AIR. As proposed by Zhang and Sekar [116], the AIR metric provides an understanding on how more restrictive a program becomes regarding allowed indirect branch targets after the introduction of a CFI policy. Through this metric, it is possible to compare and estimate the precision of different CFI implementations. The AIR computation is made using Equation 1, in which n corresponds to the number of indirect branches that exist in the program, S is the total number of valid targets allowed for an unprotected indirect branch, and $|T_i|$ is the total number of valid targets allowed for the protected indirect branch i .

$$AIR = \frac{1}{n} \sum_{j=1}^n 1 - \frac{|T_j|}{S} \quad (1)$$

On the kernel code base we used during this work, the use of a coarse-grained CFI mechanism similar to KCoFI [24], which allows indirect branches to target the beginning of any function or any instruction after a `call`, achieves an AIR of 98.64%. Although such a high value may give the impression of decent protection, it has been demonstrated that this level of permissiveness is still not enough to protect against ROP-based attacks [14, 28, 39, 40, 92].

By applying kCFI or kCFI+CGD on the same code base, an AIR of 99.99% is achieved when comparing it to the unprotected kernel. When comparing kCFI with coarse-grained CFI, the achieved AIR value is 99.93%, which is a significant improvement on the restrictiveness of indirect code paths. kCFI+CGD achieves a slightly better AIR value of 99.94% over the coarse-grained CFI AIR.

AIA. Instead of using an implicit comparison metric, Ge *et al.* [38] proposed the use of the average indirect targets allowed (AIA) metric, which captures the overall average of allowed indirect branch targets. After computing the AIA values for a program with different CFI policies, it is possible to understand their effectiveness by comparing the computed values. Equation 2, in which n is the number of indirect branches and $|T_i|$ is the number of valid targets allowed for the protected indirect branch i , is used to calculate AIA values.

$$AIA = \frac{1}{n} \sum_{j=1}^n |T_j| \quad (2)$$

The AIA values for the unprotected, coarse-grained CFI, kCFI, and kCFI+CGD kernel versions are presented in Table 1. Besides the benefits of fine-grained protection over the less restrictive policies, denoted by the three-orders-of-magnitude lower AIR value, the benefits of kCFI+CGD over kCFI also become more clear. These benefits, when analyzed through AIR, end up being hidden by the much larger magnitude of the valid branch target sets in less restrictive versions.

In kCFI+CGD, all function clones are never indirectly invocable, which positively affects permissiveness for return edges. The slight increase for calls unveils the limitation of static methods for computing the level of permissiveness of a CFI implementation. Because kCFI+CGD selectively clones functions, some indirect call

Kernel Version	Unprotected	Coarse-grained	kCFI	kCFI+CGD
AIA (all branches)	69086149	941957	680.5	545.3
AIA (only calls)	69086149	941957	60.7	62.9
AIA (only rets)	69086149	941957	952.9	769.9

Table 1: Average indirect targets allowed (AIA) metric comparison.

instructions end up being cloned, while others do not. Cloned calls are not more permissive than those of the original functions—the set of allowed targets for both is the same. Nevertheless, due to the uneven duplication of indirect calls caused by cloning, the optimization slightly increases the AIA value observed for calls. Despite this fact, as the execution of a cloned indirect call always replaces the execution of its respective original instruction, equivalent traces of kCFI and kCFI+CGD will result in the same number of executed indirect calls. As both the original and cloned calls have the same level of permissiveness, there is no harm to security, even though the metric suggests differently.

Permissiveness Comparison

The notions expressed by AIR and AIA are heavily bound to the code base being protected. As different programs will be inherently different regarding their indirect call graphs, using these metrics to compare the effectiveness of protection mechanisms requires confining the evaluation to the same code bases.

The fine-grained CFI implementation by Ge *et al.* [38] targets mainly FreeBSD, while kCFI is focused on Linux. Based on the data presented in Ge *et al.*'s paper, we can observe that the Linux version we used is 37% larger in terms of source lines of code, and has 3.3x more functions (132,859) and 4.6x more `call` instructions (809,098) than the FreeBSD version they used. While they attest that the function `printf` has approximately 5,000 possible return points in FreeBSD, `printk`, which is the corresponding one in our code base, has around 64,000. These differences imply an indirect call graph which is inherently more permissive and harder to protect, invalidating AIA and AIR comparisons between both implementations.

Ge *et al.*'s implementation incurs performance overheads that are comparable to kCFI's. To achieve this performance, it amortizes the costs introduced by instrumentation through analysis and optimizations which are also bound to the code base, such as converting indirect branches that have only one possible target into direct branches. On larger code bases, where more functions may have to be indirectly invoked, these optimization opportunities become more scarce.

6.3 Code Size Overhead

Due to the extra instrumentation instructions added in the original code, the protected binary exhibits an overhead in terms of code size. When compared to the unprotected binary, kCFI incurs a size overhead of 2%. kCFI+CGD incurs a slightly larger overhead than kCFI due to the introduction of cloned functions in the final binary. In total, 17779 functions were cloned while applying kCFI+CGD to our code base, what, in addition to the CFI instrumentation, caused an increment of 4% in code size. The observed absolute binary sizes are 705MB for the vanilla kernel, 718MB for kCFI and 732MB for kCFI+CGD. For our code base, 17779 functions were cloned when kCFI+CGD was applied.

7 Discussion

Void Function Pointer Arguments. C code allows indirect invocation of functions with mismatching prototypes through generalized arguments declared as `void` in the pointer - For example, the function `void foo(int a)` can be called through a pointer with prototype `void (void)`. Although not a good programming practice for breaking data abstractions and harming code legibility, these constructions are used to mimic polymorphism, which is not a default feature in the language. On kCFI, this issue has a second side-effect which is a deal breaker: pointers used to call functions which have mismatching prototypes will trigger a violation.

First, for identifying problematic invocations, we relied on a dynamic profiling method which was enabled by a custom violation handler capable of logging all the spots where the problems happened. After booting and stressing the kernel through the execution of LMbench and Phoronix [55,65], we used the generated information to fix the kernel code. In total, 15 prototype mismatches were observed. All of them were either fixed by changing the

function’s prototype or creating a wrapper function. A more conservative approach would be merging the clusters for the mismatching prototypes, but we rejected this solution to avoid decreasing CFI restrictiveness. As not all of the prototype mismatches existent in Linux code are intentional [61,62], this leverages a good side-capability of kCFI, which is unveiling prototype mismatches in the instrumented code.

Cache Performance. The guard instrumentation has an effect on cache schemes. This instrumentation may cause an additional L1 cache miss that certainly will be covered by an L2 cache hit if other cache levels are inclusive. On the targeted hardware platform, L1 caches are divided in code and data cache, but the same is not done in lower levels. In the worst case, when the guard dereferences a value for comparison, if the value is not on any cache level, the guard memory read is only anticipating a compulsory miss that would happen by the time the branch executes. Other possible situations are: already having the value on L1 data cache, due to this have been used by a previous guard, or already having it on lower level caches—both are less costly than the first scenario described.

Tail Call Elimination. Stack frame reuse based optimizations are a concern to CFI instrumentation as these will modify program flow. On Linux kernel we have found places prone to the application of tail call elimination, which is an optimization that replaces `call` instructions at the end of functions for direct jumps, promoting stack frame reuse by the callee function that, when finished, will return to the function underneath its caller. As the callee function will no longer return to its own call site, this optimization breaks assumptions of fine-grained CFI. Tackling this problem through dynamically verifying false-positives in the violation handler is prohibitive as a handler invocation already exceeds the optimization benefits by itself. A more reasonable approach would be merging the caller’s and callee’s entities in the CFI Map but at the cost of stretching the call graph permissiveness.

We evaluated the optimization benefits by comparing the performance of *vanilla* kernels compiled with and without it while running LMBench [65], verifying an average overhead of 5%. As this was considered low impact, especially considering that LMBench is a kernel micro-benchmark, we decided not to trade security, incorporating the overhead to kCFI’s. All performance numbers presented in this paper were measured on kCFI kernels compiled with tail call elimination disabled, while the unprotected version used for comparison had it enabled.

Loadable Kernel Modules. Although kCFI supports the use of loadable kernel modules, the analyses done to build the CFI Map are bind to the source files being compiled. Because of that, compiling modules to be loaded in a system which was previously built may break assumptions used in CFI policies, such as which functions are indirectly invocable. In these cases, CFI Map compatibility must be recovered, and this can be done through full system recompilation.

8 Related Work

As kernel exploitation through *ret2usr* methodologies was addressed by different protection technologies [21,51,70,79–81,97,111], attackers were pushed into evolving control-flow hijacking techniques for reusing code confined to the kernel address space. From these techniques, ROP and its variants [11,86] were widely employed for enabling Turing-complete computation even when launched over small code bases [12]. Although widely adopted to attack user space programs, ROP-based attacks were shown to work in kernel software [45,72].

Trying to protect the kernel against all sorts of control-flow hijacking attacks, including ROP-based variants, researchers proposed different schemes that consist of CFI implementations or approaches largely based on its idea of call-graph enforcement. The CFI concept was originally introduced by Abadi *et al.* [8], and was been largely explored for protecting user-space software [10,16,27,63,77,114–116].

DRK [36] is a solution that employs dynamic binary translation for instrumenting kernel entry points dynamically. Through this instrumentation, DRK builds a shadow memory of kernel data structures, enabling the detection of corrupted return addresses on the stack. Although this method does not require source code recompilation, shadow memory instrumentation introduces very high overheads that can reach 10x when compared to native performance.

KCoFI [24] is a coarse-grained kernel CFI implementation that, instead of computing the system’s call graph, employs a single tag for validating every indirect branch. KCoFI was built on top of a secure execution layer called Secure Virtual Architecture [25] and presents prohibitive overheads that range from 2x to 3.5x while running microbenchmark operations. Besides its elevated cost, KCoFI is also not fully capable of closing the control-flow hijacking problem in kernel software as its enforced coarse-grained policy was proved insufficient [14,28,39,40,73,74,82,84,92,105].

HyperSafe [109] employs *non-bypassable memory lockdown* and *restricted pointer indexing* to implement CFI for hypervisors. While memory lockdown protects the hypervisor code and static data from being compromised, restricted pointer indexing creates a code structure that enforces all indirect branches to target addresses present in a pre-computed destination table.

While HyperSafe is restricted to hypervisors, a similar approach is employed by Ge *et al.* [38] to implement fine-grained CFI for kernel software. As the restricted pointer indexing approach requires a pre-computation of valid target addresses, this scheme inherently breaks LKM support, which is an important feature of most modern kernels. Besides that, achieving good performance in this implementation requires amortizing the costs introduced by execution indirections, what is done through code optimizations that may not be applicable and that are more scarce on larger code bases.

PaX [100] is a Linux hardening patch set that protects the system against many different kinds of attack, including some variants of control-flow hijacking attacks. *Return Address Protection* [101] is one of the features available in PaX and is conceptually based on the XOR canary approach used by StackGuard [23]. This feature encrypts return addresses and keeps the decryption key, which is a XOR cookie, in a reserved general-use register. Return address encryption based mechanisms have been proposed before [75] and, while they certainly impose an obstacle to attackers, the whole scheme security relies on the secrecy of the cookie, which compromises the protection if leaked or brute-forced. RAP also implements tag verification to validate indirect branches but its enforced CFG is fully based on strict prototype matching, not taking measures to solve transitivity relaxations or to prevent clusterization of functions that will never be indirectly invoked.

Recently, Intel released technology specifications for a new instruction set specification called *Control Flow Enforcement* (CET) [22] that extends `call` and `ret` instructions to inherently use of a shadow stack structure to enforce valid returns. CET also introduces new instructions to be used for marking valid targets for indirect calls. Although the introduced shadow stack is a significant feature, the model used for validating indirect calls is based on weak coarse-grained policies [14, 28, 39, 40, 92]. CET is not yet available in the market, and little is known regarding its compatibility with OSes, programming language features, and code optimizations.

Researchers shown that fine-grained CFI implementations may also be vulnerable to attacks [13, 32]. These attacks exploit the permissiveness of the computed call graphs in which the CFI policy is based, redirecting control-flow through paths which are considered valid but in a malicious way. Even though no study exists about how often the requirements for these attacks are found in the kernel context, kCFI raises the bar for these attacks by decreasing the call graph permissiveness through applying the CGD technique described in Section 4.2.

9 Conclusion

We presented kCFI, a fine-grained tag-based CFI implementation capable of supporting the Linux kernel and protecting it against modern control-flow hijacking techniques, including ROP attacks. kCFI works by instrumenting the kernel source code with indirect branch assertions that verify the validity of a control-transfer before its execution. kCFI's design takes advantage of traits in the x86-64 architecture, introducing only low-cost memory operations and making use of instructions with negligible overhead to mark code. Unlike some previous approaches, kCFI achieves its goals without using restricted pointer indexing or converting indirect branches into direct ones. For this reason, kCFI does not harm any system features and reliably supports LKMs.

Availability

kCFI is available as an open-source project at: <https://github.com/kcfi/>

References

- [1] Control Flow Guard. [https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx).
- [2] Control Flow Integrity. <http://clang.llvm.org/docs/ControlFlowIntegrity.html>.
- [3] CVE-2010-3437, September 2010.
- [4] CVE-2013-6282, October 2013.
- [5] CVE-2015-3036, April 2015.
- [6] CVE-2015-3290, April 2015.
- [7] Control-flow Enforcement Technology Preview, 2016. <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>.

- [8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. of CCS*, pages 340–353, 2005.
- [9] P. Argyroudis. Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation. In *Black Hat USA*, 2010.
- [10] T. Bletsch, X. Jiang, and V. Freeh. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proc. of ACSAC*, pages 353–362, 2011.
- [11] 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, ASIACCS '11*, pages 30–40, New York, NY, USA, 2011. ACM.
- [12] 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, CCS '08*, pages 27–38, New York, NY, USA, 2008. ACM.
- [13] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proc. of USENIX Sec*, pages 161–176, 2015.
- [14] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. of USENIX Sec*, pages 385–399, 2014.
- [15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proc. of CCS*, pages 559–572, 2010.
- [16] 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. of NDSS*, 2014.
- [17] Common Vulnerabilities and Exposures. CVE-2010-3301, September 2010.
- [18] Common Vulnerabilities and Exposures. CVE-2010-3904, October 2010.
- [19] J. Corbet. Virtual Memory I: the problem. <http://lwn.net/Articles/75174/>, March 2004.
- [20] J. Corbet. An updated guide to debugfs. <http://lwn.net/Articles/334546/>, May 2009.
- [21] J. Corbet. Supervisor mode access prevention. <http://lwn.net/Articles/517475/>, October 2012.
- [22] I. Corporation. Control-flow enforcement technology preview. "<https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>", 6 2016. [Online; accessed 7-August-2016].
- [23] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7, SSYM'98*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.
- [24] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proc. of IEEE S&P*, pages 292–307, 2014.
- [25] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, pages 351–366, New York, NY, USA, 2007. ACM.
- [26] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15*, pages 555–566, New York, NY, USA, 2015. ACM.
- [27] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proc. of NDSS*, 2012.
- [28] 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. of USENIX Sec*, pages 401–416, 2014.

- [29] S. Designer. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>, August 1997.
- [30] J. Edge. Kernel address space layout randomization. <http://lwn.net/Articles/569635/>, October 2013.
- [31] S. Eren. Smashing The Kernel Stack For Fun And Profit. *Phrack*, 6(60), December 2002.
- [32] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proc. of CCS*, pages 901–913, 2015.
- [33] Exploit Database. EBD-20201, August 2012.
- [34] Exploit Database. EBD-31346, February 2014.
- [35] Exploit Database. EBD-33516, May 2014.
- [36] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 135–146, New York, NY, USA, 2012. ACM.
- [37] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proc. of ACSAC*, pages 396–405, 2014.
- [38] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy 2016*, Euro S&P, Washington, USA, 2016. IEEE Computer Society.
- [39] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.
- [40] 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. of USENIX Sec*, pages 417–432, 2014.
- [41] Google. Android. <http://www.android.com>.
- [42] Google. Chromium OS. <http://www.chromium.org/chromium-os>.
- [43] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection Without Loose Ends. In *Proc. of ACSAC*, pages 341–350, 2015.
- [44] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec*, pages 384–398, 2009.
- [45] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM’09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.
- [46] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proc. of NDSS*, 2014.
- [47] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. of IEEE S&P*, pages 2–12, 1984.
- [48] S. Keil and C. Kolbitsch. Kernel-mode exploits primer. Technical report, International Secure Systems Lab (isecLAB), November 2007.
- [49] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. of USENIX Sec*, pages 957–972, 2014.
- [50] V. P. Kemerlis, G. Portokalidis, E. Athanasopoulos, and A. D. Keromytis. kGuard: Lightweight Kernel Protection. *USENIX ;login.*, 37(6):7–14, December 2012.
- [51] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX Sec*, pages 459–474, 2012.

- [52] T. J. Killian. Processes as Files. In *Proc. of USENIX Summer*, pages 203–207, 1984.
- [53] A. Kleen. Memory Layout on amd64 Linux. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, July 2004.
- [54] G. Kroah-Hartman. udev – A Userspace Implementation of devfs. In *Proc. of OLS*, pages 263–271, 2003.
- [55] M. Larabel and M. Tippet. Phoronix Test Suite. <http://www.phoronix-test-suite.com/>, 2011. [Online; acesso 07/08/2016].
- [56] M. Larkin. Kernel W^X Improvements In OpenBSD. In *Hackfest*, 2015.
- [57] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with “return-less” kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.
- [58] S. Liakh. NX protection for kernel data. <http://lwn.net/Articles/342266/>, July 2009.
- [59] LLVMLinux. <http://llvm.linuxfoundation.org/>. [Online; acesso 07/08/2016].
- [60] Linux Foundation. Tizen. <https://www.tizen.org>.
- [61] Linux Kernel Mailing List. [PATCH] ACPI: fix acpi_debugfs_init prototype. <https://lkml.org/lkml/2015/8/1/72>.
- [62] Linux Kernel Mailing List. [tip:x86/mm] module: fix () used as prototype in include/linux/module.h. <https://lkml.org/lkml/2010/2/17/250>.
- [63] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 941–951, New York, NY, USA, 2015. ACM.
- [64] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*, chapter Overview of the FreeBSD Virtual-Memory System, pages 227–230. Pearson Education, 2nd edition, 2015.
- [65] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
- [66] Microsoft. Microsoft Windows. <http://windows.microsoft.com>.
- [67] Microsoft Developer Network. Memory Limits for Windows and Windows Server Releases. [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx).
- [68] Microsoft Developer Network. Virtual Address Space. <https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912%28v=vs.85%29.aspx>.
- [69] I. Molnar. 4G/4G split on x86, 64 GB RAM (and more) support. <http://lwn.net/Articles/39283/>, July 2003.
- [70] J. Morse. arm64: kernel: Add support for Privileged Access Never. <https://lwn.net/Articles/651614/>, July 2015.
- [71] National Vulnerability Database. Kernel Vulnerabilities. https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&query=kernel&cvss_version=3, October 2016.
- [72] V. Nikolenko. Linux kernel rop - ropping your way to root (part 1). [https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---\(Part-1\)/](https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/). [Online; accessed 16-August-2016].
- [73] B. Niu and G. Tan. Modular Control-flow Integrity. In *Proc. of PLDI*, pages 577–587, 2014.
- [74] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *Proc. of CCS*, pages 914–926, 2015.

- [75] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10*, pages 49–58, New York, NY, USA, 2010. ACM.
- [76] T. L. K. Organization. Linux. <https://www.kernel.org>.
- [77] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. of USENIX Sec*, pages 447–462, 2013.
- [78] PaX Team. Address Space Layout Randomization (ASLR), 2003. <http://pax.grsecurity.net/docs/aslr.txt>.
- [79] PaX Team. UDEREF/i386. <http://grsecurity.net/~spender/uderef.txt>, April 2007.
- [80] PaX Team. UDEREF/amd64. <https://goo.gl/iPu0VZ>, April 2010.
- [81] PaX Team. Better kernels with GCC plugins. <http://lwn.net/Articles/461811/>, October 2011.
- [82] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proc. of DIMVA*, pages 144–164, 2015.
- [83] E. Perla and M. Oldani. *A Guide To Kernel Exploitation: Attacking the Core*, chapter Stairway to Successful Kernel Exploitation, pages 47–99. Elsevier, 2010.
- [84] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proc. of ACSAC*, pages 309–318, 2013.
- [85] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of NDSS*, 2015.
- [86] M. Prandini and M. Ramilli. Return-oriented programming. *Security Privacy, IEEE*, 10(6):84–87, Nov 2012.
- [87] PUC-Rio. The Programming Language Lua. <http://www.lua.org>.
- [88] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.
- [89] D. Rosenberg. `kpctr_restrict` for hiding kernel pointers. <http://lwn.net/Articles/420403/>, December 2010.
- [90] M. D. Schroeder and J. H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, March 1972.
- [91] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of IEEE S&P*, pages 745–762, 2015.
- [92] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proc. of RAID*, pages 88–108, 2014.
- [93] SecurityFocus. Linux Kernel 'perf_counter_open()' Local Buffer Overflow Vulnerability, September 2009.
- [94] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of CCS*, pages 552–61, 2007.
- [95] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security, CCS '04*, pages 298–307, New York, NY, USA, 2004. ACM.
- [96] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13*, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

- [97] B. Spengler. Recent ARM security improvements. <https://goo.gl/H8HkwD>, February 2013.
- [98] B. Spengler. Enlightenment Linux Kernel Exploitation Framework. <https://grsecurity.net/~spender/exploits/enlightenment.tgz>, December 2014.
- [99] sqrkkyu and twiz. Attacking the Core: Kernel Exploiting Notes. *Phrack*, 6(64), May 2007.
- [100] P. Team. Pax. "<https://pax.grsecurity.net>", Apr 2013. [Online; accessed 2-August-2016].
- [101] P. Team. Rap: Rip rop. "<https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-R0P.pdf>", 10 2015. [Online; accessed 7-August-2016].
- [102] The FreeBSD Foundation. FreeBSD. <https://www.freebsd.org>.
- [103] The NetBSD Foundation. NetBSD. <http://www.netbsd.org>.
- [104] The OpenBSD Foundation. OpenBSD. <http://www.openbsd.org>.
- [105] 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. of USENIX Sec*, pages 941–955, 2014.
- [106] A. van de Ven. Debug option to write-protect rodata: the write protect logic and config option. <http://lkm1.indiana.edu/hypermil/linux/kernel/0511.0/2165.html>, November 2005.
- [107] A. van de Ven. Add `-fstack-protector` support to the kernel. <http://lwn.net/Articles/193307/>, July 2006.
- [108] 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 *Proc. of USENIX Sec*, pages 813–828, 2014.
- [109] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, May 2010.
- [110] Xst3nZ. Windows Kernel Exploitation Basics. <http://poppopret.blogspot.com/2011/07/windows-kernel-exploitation-basics-part.html>, July 2011.
- [111] F. Yu. Enable/Disable Supervisor Mode Execution Protection. <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679>, May 2011.
- [112] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proc. of CCS*, pages 29–40, 2011.
- [113] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. of NDSS*, 2015.
- [114] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres. Protecting Function Pointers in Binary. In *Proc. of ASIACCS*, pages 487–492, 2013.
- [115] 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. of IEEE S&P*, pages 559–573, 2013.
- [116] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX Sec*, pages 337–352, 2013.