Per-Input Control-Flow Integrity

Ben Niu
Lehigh University
19 Memorial Dr West
Bethlehem, PA, 18015
ben210@lehigh.edu

Gang Tan
Lehigh University
19 Memorial Dr West
Bethlehem, PA, 18015
gtan@cse.lehigh.edu

ABSTRACT

Control-Flow Integrity (CFI) is an effective approach to mitigating control-flow hijacking attacks. Conventional CFI techniques statically extract a control-flow graph (CFG) from a program and instrument the program to enforce that CFG. The statically generated CFG includes all edges for all possible inputs; however, for a concrete input, the CFG may include many unnecessary edges.

We present Per-Input Control-Flow Integrity (PICFI or πCFI), which is a new CFI technique that can enforce a CFG computed for each concrete input. πCFI starts executing a program with the empty CFG and lets the program itself lazily add edges to the enforced CFG if such edges are required for the concrete input. The edge addition is performed by πCFI-inserted instrumentation code. To prevent attackers from arbitrarily adding edges, πCFI uses a statically computed all-input CFG to constrain what edges can be added at runtime. To minimize performance overhead, operations for adding edges are designed to be idempotent, so they can be patched to no-ops after their first execution. As our evaluation shows, πCFI provides better security than conventional fine-grained CFI with comparable performance overhead.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

Keywords

Control-Flow Integrity; Dynamic CFI

1. INTRODUCTION

Modern software exploitation techniques such as Return-Oriented Programming (ROP [27]) rely on hijacking the control flow of a victim program to abnormally execute dangerous system calls (e.g., `mprotect` and `execve`). Although mainstream operating systems (i.e., Windows, Linux and OSX) have deployed mitigation methods such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), control-flow hijacking is still one of the largest threats to software security. Besides those already-in-use mitigation methods, another effective way of defending against control-flow hijacking attacks is Control-Flow Integrity (CFI, [1]). In its conventional form, CFI statically computes a Control-Flow Graph (CFG) and instruments the binary code by adding checks before indirect branches (i.e., indirect calls, indirect jumps, and returns). These checks ensure any control transfer during execution never deviates from the CFG, even under attacks.

Despite effectiveness, not every CFI implementation provides the same level of protection, since CFI's security depends on the precision of the enforced CFG. The more precise the CFG is, the less choices attackers have when redirecting the control flow, the more security a particular CFI implementation provides. Coarse-grained CFI (e.g., CCFIR [34], binCFI [35], and kBouncer [21]) enforces a coarse-grained CFG in which there are a few equivalence classes of target addresses (or even just one equivalence class) and an indirect branch is allowed to target one of the equivalence classes. For example, binCFI allows return instructions to return to all possible return addresses. Unfortunately, the precision of coarse-grained CFI is too low and it is still possible to mount ROP attacks on coarse-grained CFI, as shown in recent work [13, 10, 8]. Fine-grained CFI enforces a much higher-precision CFG; each indirect branch can have its own set of target addresses. For instance, MCFI [19] and forward-edge CFI [29] are compiler-based frameworks that build fine-grained CFGs from source code using high-level type information.

Whether it be coarse-grained or fine-grained CFI, a fundamental limitation of previous CFI methods is that the enforced CFGs are computed by static analysis, which has to consider *all possible program inputs*. Consequently, the precision of a statically computed CFG cannot be better than an "ideal" CFG, which is the minimal CFG when considering all program inputs. Computing such "ideal" CFGs is in general intractable and static analysis has to over-approximate. More importantly, even an "ideal" CFG includes unnecessary edges for *a concrete program input*. Therefore, our idea is to explore whether it is possible to generate a CFG for each concrete input and enforce it at runtime for that input alone. Intuitively, the per-input CFG should have a better precision than a statically computed, all-input CFG.

In this paper, we present Per-Input Control-Flow Integrity (PICFI, or πCFI), a general CFI method for generating and enforcing per-input CFGs. Since it is impossible to enumerate all inputs of a program, computing the CFG for each input and storing all per-input CFGs are infeasible. Instead, we adopt the following approach: we start a program with the empty CFG and let the program itself lazily compute the CFG on the fly. One idea of computing the CFG lazily is to add edges to the CFG at runtime, before indirect branches need those edges. In this way, the per-input CFG generation problem becomes feasible: for an arbitrary input, the dynamically gen-

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813644.

erated and enforced CFG is equivalent to what should have been computed prior to the execution.

However, two challenges still remain to be addressed. First, since edge addition is issued by untrusted code, how to prevent it from arbitrarily adding edges? πCFI should be able to identify those edges that shall never be added. To address this challenge, πCFI reuses previous CFI methods to first compute an all-input CFG statically. Specifically, our πCFI implementation builds on MCFI and RockJIT [20] for all-input C/C++ CFG generation using source-level type information and class hierarchies, but it could be based on any previous CFI method. Then πCFI starts running the program with the empty CFG being enforced. At runtime, the program adds edges on the fly, but πCFI disallows addition of any edge not included in the static, all-input CFG. In other words, the all-input CFG serves as the upper bound for what edges can be added to the enforced CFG during runtime.

The second challenge is how to achieve small performance overhead for enforcing πCFI? For each indirect branch, there is a need to add the necessary edge into the CFG. This can be achieved by code instrumentation; that is, by inserting extra code that adds edges into the original program. However, such instrumentation can be costly since every time the indirect branch gets executed, the edge-addition operation needs to be performed. πCFI adopts a performance optimization technique, with some loss of CFG precision. This technique turns edge addition to *address activation*. In particular, instead of directly adding edges, πCFI activates target addresses. Activating an address essentially adds all edges with that address as the target into the currently enforced CFG. The benefit of using address activation operations is that they can be made idempotent operations with some careful design. With idempotent operations, we can safely patch them to no-ops after their first execution, minimizing performance overhead.

In summary, we highlight our contributions below:

- We propose πCFI, a general CFI method for generating and enforcing per-input CFGs. Our experiments show that πCFI eliminates many unnecessary edges in its enforced CFGs. For SPECCPU2006 benchmarks, on average the number of edges in the enforced CFGs are only about 10.4% of the number of edges in static, all-input CFGs. Moreover, πCFI's runtime overhead is small; only 3.2% on average for SPECCPU2006 benchmarks.
- We propose techniques that make the per-input CFI idea efficient and secure, including lazy CFG computation, idempotent address activation, and secure code patching.
- We have built a framework to harden applications with πCFI, and evaluated πCFI's security and performance with respect to SPECCPU2006, the Google V8 JavaScript engine, and the Nginx HTTP server. Our implementation is publicly available for download on `https://github.com/mcfi`.

The remainder of this paper is organized as follows. We introduce related work in Sec. 2 before providing an overview of πCFI in Sec. 3. We describe technical details of πCFI in Sec. 4 and the implementation in Sec. 5. We analyze πCFI's security in Sec. 6 and performance in Sec. 7. Finally in Sec. 8 and 9 we discuss future work and draw conclusions.

2. RELATED WORK

CFI [1] extracts a Control-Flow Graph (CFG) from a program and instruments the program by adding checks before indirect branch instructions. During runtime, before any indirect branch is executed, the inserted checks query the CFG to ensure that only targets whitelisted in the CFG are allowed. As a result, CFI guarantees that any runtime control-flow transfer must correspond to an edge in the CFG. The converse does not always hold, as a CFG is a static over-approximation of a program's runtime control flow. A program can have many CFGs; some are more precise than others.

For each CFG of a program, indirect branches and their targets can be partitioned into *equivalence classes*. Indirect branches can target any indirect branch targets in the same equivalence class, but none in other equivalence classes. Different CFI techniques support different numbers of equivalence classes. In general, CFI techniques in the literature can be classified into two categories: *coarse-grained* CFI and *fine-grained* CFI, depending on their support for equivalence classes.

Coarse-grained CFI supports *program-agnostic* number of equivalence classes, which is usually no more than three. In coarse-grained CFGs, typically each kind of indirect branches is allowed to target one equivalence class. For instance, in binCFI return instructions are allowed to target all return addresses, which are addresses following call instructions. Example coarse-grained CFI techniques include PittSFIeld [16], NaCl [32, 26], CCFIR [34], binCFI [35], and MIP [18]. The major benefit of coarse-grained CFI is that coarse-grained CFGs are easier to build, even without access to source code (e.g., [17]). But on the down side, the coarse-grained CFGs are too permissive so that it is still possible to mount attacks in general, as demonstrated in recent work [13, 10, 8].

Fine-grained CFI supports *program-dependent* number of equivalence classes. Each indirect branch can have its own target set. Example fine-grained CFI approaches include several systems [12, 31, 2, 9, 23]. However, limited by their CFI enforcement mechanisms, none of them supports modularity (dynamic code linking or Just-In-Time compilation). Tice *et al.* [29] proposed an approach to fine-grained CFI with (partial) modularity support. However, it does not protect return instructions, and its modularity support introduces time windows for attacks during dynamic module linking. MCFI [19] is the first fine-grained CFI technique that supports dynamic code linking, and its follow-up work RockJIT [20] extends MCFI to support fine-grained CFG generation for C++ and secure Just-In-Time (JIT) compilation. (In the remainder of this paper, We use MCFI to reference the combined work in [19] and [20].) Lockdown [22] is another fine-grained CFI enforcement system, which can work on stripped binaries without access to source code. However, its execution performance overhead is higher than MCFI because its implementation is based on dynamic binary translation.

None of the above mentioned CFI techniques supports per-input CFGs; they provide less security than πCFI. Independently and concurrently with πCFI, HAFIX [4] enforces per-input CFGs using specially built hardware. However, compared to πCFI's support for all indirect branches, HAFIX only supports per-input CFGs with respect to returns. HAFIX also lacks support for multi-threaded programs, which are supported by πCFI. In addition, πCFI is a pure software-based technique that can run on existing commodity hardware, which is more deployable than HAFIX that modifies the hardware.

Systems such as XFI [12] protect the integrity of the stack by using a shadow stack. It ensures that a return instruction always returns to its actual runtime call site. πCFI's protection on return instructions falls between conventional CFI and the shadow-stack defense: it ensures a return instruction in a function can return to only those call sites that have so far called the function. πCFI, on the other hand, better protects other indirect branches (e.g., indirect calls) and is compatible with unconventional control flow mechanisms such as exception handling. A more detailed comparison will be offered in the next section.

Code-Pointer Integrity (CPI [15]) is a recent system that isolates all data related to code pointers into a protected safe memory region and thus can mitigate control-flow hijacking attacks. It is also a compiler-based framework and has low performance overhead. However, it relies on a whole-program analysis that separates code-pointer data and the rest and lacks modularity support. Furthermore, CPI does not directly enforce a control-flow graph. The control-flow graph provided by CFI methods such as πCFI is valuable to other software-protection mechanisms because they can use it to perform static-analysis based optimization and verification [33].

3. πCFI MOTIVATION AND OVERVIEW

Before introducing the detailed system design of πCFI, in this section we present an overview, including its threat model, some terminology, motivation for per-input CFGs, and the benefits of address activation.

3.1 Threat model

πCFI protects user-level applications. Its trusted computing base includes the following components: the underlying software-hardware stack (including the CPU, the virtual machine if there is one, and the operating system); πCFI's LLVM-based compilation toolchain; and πCFI's runtime. In the entire execution of a πCFI-protected program, the runtime maintains the code and data separation: those virtual memory pages containing code (including dynamically generated code) and read-only data are non-writable, and those virtual pages containing data are non-executable. Similar to other CFI work, πCFI assumes that attackers have full control over all writable virtual memory pages allocated to the application. An attacker can modify any location of those pages between two consecutive instructions.

3.2 Terminology

We introduce some terminology that will make the following discussion more convenient. Conceptually, a CFI method involves two kinds of control-flow graphs:

- A static, all-input CFG. This is typically computed by static analysis. We call this the *Static CFG*, abbreviated to SCFG.
- The CFG that is currently enforced. Checks are inserted before indirect branches to consult this CFG to decide whether indirect branches are allowed. We call this the *Enforced CFG*, abbreviated to ECFG.

In previous CFI methods, SCFG = ECFG, and we call them *conventional CFI*. In πCFI, SCFG $\supseteq$ ECFG. When a program starts in πCFI, the ECFG is the empty graph. As the program runs, the ECFG grows, but πCFI uses the SCFG to upper bound the growth; that is, the ECFG is always a subgraph of the SCFG.

These two kinds of CFGs could be represented as two separate data structures, but πCFI uses one single data structure to represent both: the SCFG is represented as two tables; the ECFG is represented by marking the SCFG with special bits, which tell what addresses have been activated. We will discuss the representation in detail in the implementation section (Sec. 5).

3.3 Motivation for per-input CFGs

πCFI's Enforced CFG (ECFG) is computed for each specific input. We next use a toy C program listed in Figure 1 to illustrate its high-level idea and security benefits. The main function in the program has an if branch, whose condition depends on the number of command-line arguments. Assume that the number of command-line arguments is greater than or equal to two in a particular production environment. The main function invokes the foo function (whose code is omitted) to handle user inputs. Let us assume that foo's code has a stack-overflow vulnerability that enables attackers to control its return target. Apparently, this vulnerability can be easily exploited to hijack the control flow of this program. (For simplicity, we ignore ASLR and stack canaries in our discussion since they are orthogonal defense mechanisms to CFI.)

```
void foo(void) {
  /* We omit code that handles user inputs. The
     code contains a stack buffer overflow so
     that attackers can control the following
     return instruction's target. */
  ...
  return;
}
int main(int argc, char *argv[]) {
  if (argc < 2) {
    foo();
    L1:
    ... /* irrelevant code, omitted */
    execve(...); /* arguments omitted */
  } else {
    foo();
    L2: ...
  }
}
```

Figure 1: A motivating example for per-input CFGs.

With conventional CFI protection, which enforces a CFG for all inputs, this particular program is still vulnerable. Notice that the main function invokes foo at two different places. As a result, both L1 and L2 are possible return addresses for foo. In conventional CFI, foo's return is always allowed to target both addresses. Therefore, even if the program executes only the else branch when deployed, attackers can still control foo's return and redirect it to L1. With appropriate data manipulation, the attacker might execute the following execve with arbitrary arguments.

With πCFI, such an attack can be prevented. One possible instrumentation method is shown in Figure 2 so that the program can add its required edges during execution. (Instead of edge addition, πCFI actually uses address activation, which will be discussed later.) The program is started with the empty ECFG. At runtime, the else branch will be executed, but right before foo is called at line 20, the edge from foo's return to L2 is added (by calling πCFI's trusted runtime at line 19). When foo returns, it is only allowed to target L2, not L1, as no such an edge has been added to the ECFG.

We note that the example in Figure 1 can also be protected by defenses that protect the stack through a shadow stack. For instance XFI [12] adopts the shadow-stack defense to protect return addresses. This ensures that a function returns to the caller that called it. As a result, the return instruction in foo can return only to L2 when it is called in the else branch. In comparison, πCFI's protection on return instructions is weaker: it ensures a return instruction in a function can return to only those call sites that have so far called the function. On the other hand, πCFI offers a number of benefits than the shadow-stack approach. First, it provides stronger protection for indirect calls. For instance, if in an SCFG an indirect call is allowed to target two functions, say f1 and f2, but in one code path only f1's address is taken, then the indirect call will be disallowed to target f2 in πCFI. XFI, as

```
1  void foo(void) {
2    /* We omit code that handles user inputs. The
3       code contains a stack buffer overflow so
4       that attackers can control the following
5       return instruction's target. */
6    ...
7    return;
8  }
9  int main(int argc, char *argv[]) {
10   if (argc < 2) {
11     /* connect foo's return to L1 */
12     add_edge(foo, L1); /* Instrumentation */
13     foo();
14     L1:
15     ... /* irrelevant code, omitted */
16     execve(...); /* arguments omitted */
17   } else {
18     /* connect foo's return to L2 */
19     add_edge(foo, L2); /* Instrumentation */
20     foo();
21     L2: ...
22   }
23 }
```

Figure 2: Edge-addition instrumentation for the motivating example.

it stands, allows an indirect call to target any address according to the static CFG and cannot restrict the set of targets per a specific input as πCFI does. Second, the shadow-stack defense traditionally has compatibility issues with code that uses unconventional control-transfer mechanisms including setjmp/longjmp, exceptions, and continuations since they do not follow the rigid call-return matching paradigm. πCFI offers the compatibility advantage because it can be parametrized by any SCFG and is compatible with code that uses unconventional control-transfer mechanisms.

However, since πCFI does not perform address deactivation (except in rare situations when a code module is explicitly unloaded), one worry is that most of the time its ECFG grows along with the program execution. In theory, an attacker might use some malicious input to trigger the activation of all targets in an SCFG, in which case πCFI falls back to conventional CFI. This is especially a concern for a long running program that keeps taking inputs, such as a web server or a web browser. However, we believe πCFI offers benefits even for such kind of programs, for the following reasons:

- An attacker would need to find a set of inputs that can trigger the activation of all targets of her/his interest; this is essentially asking the attacker to solve the code coverage problem, a traditionally hard problem in software testing.

- Our preliminary experiments (presented in Sec. 6) suggest that the number of edges in an ECFG stabilizes to a small percentage of the total number of edges in an SCFG even for long running programs that continuously take normal user inputs. We believe this is due to several factors. First, a typical application includes a large amount of error-handling code, which will not be run in normal program execution. For instance, Saha *et al.* [24] found that 48% of Linux 2.6.34 driver code is found in functions that handle at least one error and in general systems software contains around 43% of the code in functions that contain multiple blocks of error-handling code. Second, an application may contain code that handle different configurations (like the motivating example) of execution environments. It is generally hard for a static analysis to construct a per-configuration CFG as it has to consider features such as environment variables and C macros. Finally, static analysis has to over-approximate when constructing static CFGs. As a result, many dynamically unreachable edges are included. For instance, static analysis may fail to recognize dead code in the application and allow indirect branches to target addresses in the dead code. This is especially the case for functions in library code.

- A long running program that continuously takes user inputs typically forks new processes or pre-forks a pool of processes for handling new inputs. For instance, web servers such as Apache and Nginx pre-fork a process pool for handling web requests. In πCFI, the CFG growth of a child process is independent of the CFG growth of the parent process. This setup limits the CFG growth of such programs.

3.4 From edge addition to address activation

The simple instrumentation shown in Figure 2 has performance problems: each time `foo` is invoked, `add_edge` is also invoked. Although we can use static analysis to eliminate redundant edge-addition calls (e.g., it might be possible to hoist such calls outside a loop), it would be hard to minimize such instrumentation code. Instead, we propose an alternative approach.

We design every operation that modifies the ECFG to be *idempotent* and eliminate it by patching it to no-ops after its first execution. An idempotent operation is designed so that the effect of performing it twice is the same as the effect of performing it only once. Therefore, after the first time, there is no need to perform it again. For example, the operation on line 19 in Figure 2 is idempotent: it transfers the control to the trusted runtime, and the runtime adds an edge from `foo`'s return to `L2` to the CFG. Before the runtime returns, it can patch the code at line 19 with no-ops to reduce any subsequent execution's cost.[1] Furthermore, as we will explain, using idempotent operations is also important for code synchronization when performing online code patching in multi-threaded applications running on multi-core architectures.

However, how to make every edge addition idempotent? Consider an example of an indirect call. Before the indirect call, we could add an edge addition to register the edge required to execute the call. However, this operation is not idempotent, because the indirect call may have a different target next time it is invoked. One solution is to use an operation that adds all possible edges for the indirect call according to the SCFG. This operation is idempotent, but is incompatible with dynamic linking, during which the SCFG itself changes and new targets for the indirect call may be added.

Our solution is to turn edge addition to address activation of statically known addresses to enable idempotence. In general, we observe that *only if an indirect branch's target address is activated, can the address be reachable by the indirect branch.* Activating an address has the same effect as adding all edges that target the address from the current (and future) SCFG to the current (and future) ECFG. Activating a statically known address is idempotent, as activating the same address twice has the same effect as activating it only once.

4. πCFI SYSTEM DESIGN

In this section, we discuss the detailed system design of πCFI, including how it achieves secure online code patching, how it ac-

[1] The edge addition happens only once in the code of Figure 2, but in other examples such an operation may be executed multiple times, for instance, when it is in a loop.

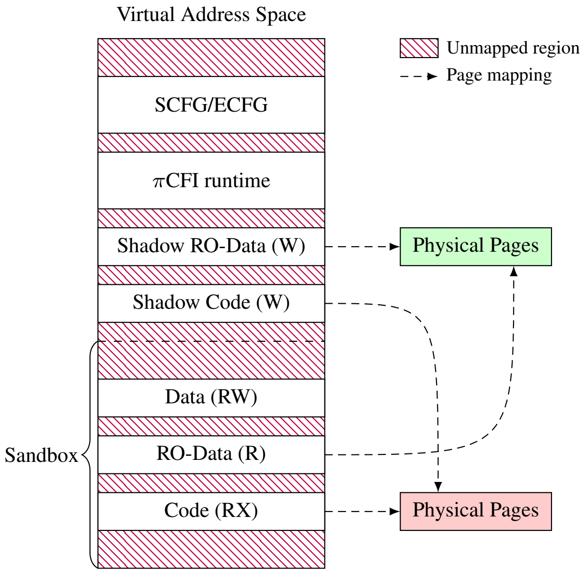

Figure 3: Memory layout of πCFI. "R", "W" and "X" appearing in parentheses denote the Readable, Writable, and eXecutable memory page permissions, respectively. The "RO-" prefix means Read-Only.

tivates addresses for each kind of indirect branch target addresses, and how it is made compatible with typical software features.

4.1 Secure code patching

Idempotent address-activation operations allow πCFI to patch the operations with no-ops after their first execution, but the patching should be securely performed. Online code patching typically implies granting the writable permission to code pages, which enables code-injection attacks. To avoid such risks, we adopt the approach of RockJIT [20] and NaCl-JIT [3] that securely handles JITted code manipulation and generalize it to patching regular code (i.e., non-JITted code).

Figure 3 shows the memory layout of an application protected with πCFI. The application should have been compiled and instrumented by πCFI's compilation toolchain. The application and all its instrumented libraries are loaded into a sandbox created by the πCFI runtime. The sandbox can be realized using Software-based Fault Isolation (SFI [30]) or some hardware support. For example, we can instrument all memory writes to ensure their targets always stay within the sandbox. Code in the sandbox cannot arbitrarily execute or write memory pages outside the sandbox, but has to invoke trampolines provided by the πCFI runtime; these trampolines allow the untrusted code to escape the sandbox safely. The runtime also maintains the invariant that no memory pages in the sandbox are writable and executable simultaneously, at any time. In addition, the runtime guarantees that read-only data, such as jump tables, are not writable. The πCFI runtime and the encoded SCFG/ECFG stay outside the sandbox. The encoded SCFG/ECFG is read-only from the application's perspective, but writable by the runtime.

To enable secure patching, πCFI's runtime allocates another set of writable virtual memory pages, called *shadow code pages*, outside the sandbox and maps these pages to exactly the same physical pages as the application's code pages inside the sandbox. The shadow code pages are writable by the runtime, but cannot be modified by the application since those pages are outside the sandbox. In this way, πCFI maintains the invariant that no memory pages in the sandbox are writable and executable at the same time. More importantly, the πCFI runtime can securely perform code patches on the shadow code pages and these changes are synchronously reflected in the application's code pages since they are mapped to the same physical pages.

Figure 3 also shows parallel mapping of the read-only data. They contain runtime-adjustable read-only data, especially the GOT.PLT data in Linux. The PLT (Procedure Linkage Table) contains a list of entries that contain glue code emitted by the compiler to support dynamic linking. Code in the PLT entries uses target addresses stored in the GOT.PLT table (GOT is short for Global Offset Table). The GOT.PLT table is adjusted during runtime by the linker to dynamically link modules. However, security weakness also results from the GOT.PLT table's writability, as demonstrated by an attack [10]. To address this security concern, πCFI sets the GOT.PLT table to be always read-only inside the sandbox and creates outside the sandbox a shadow GOT.PLT table, which is mapped to the same physical pages as the in-sandbox GOT.PLT table. All changes to the GOT.PLT table are therefore checked and performed by the πCFI runtime, which ensures that each entry's value is the address of either the dynamic linker or the address of a function whose name is the same as the corresponding PLT entry's name.

4.2 Address activation

πCFI dynamically activates indirect branch targets. When a target address is submitted to πCFI's runtime for activation, it consults the encoded SCFG to check if the address is a valid target address; if so, the runtime activates the address (by enabling it in the ECFG) so that future indirect branches can jump to it.

For each target address, there is a time window during which that target can be activated—from the beginning of program execution to immediately before the target is first used in an indirect branch; in the case when a target is never used in a program run, the time window is from the beginning of program execution to infinity. One way to think of conventional CFI is to view it as an approach that eagerly activates all target addresses at the beginning of program execution. πCFI, on the other hand, wants to delay address activation as late as possible to improve security. One natural approach would be to always activate a target immediately before its first use. This approach, however, does not take into account other constraints, which are discussed as follows:

- *Idempotence.* As we mentioned before, for efficiency we want every address-activation operation to be idempotent so that we can patch it to no-ops after its first execution. This constraint implies that not every address activation can happen immediately before its first use. We previously discussed the indirect-call example: if we insert an address-activation operation for the actual target immediately before the indirect call, that operation is not idempotent because the target might be different next time the indirect call is invoked.

- *Atomic code updates.* It is tricky to perform online code patching on modern multi-core processors. If some code update by a thread is not atomic, then it is possible for another thread to even see corrupted instructions. Therefore, a πCFI patch operation must be atomic, which means that any hardware thread should either observe the address-activation operation before the patch or the no-ops after the patch. Fortunately, x86 CPUs manufactured by both Intel and AMD support atomic instruction stream changes if the change is of eight bytes and made to an eight-byte aligned memory address, as confirmed by Ansel *et al.* [3]. We take advantage of this hardware support to implement πCFI's instrumentation and patching. It is important to stress that it is possible that the code in memory has been atomically patched by one thread, but the code cache for a different hardware thread might still contain the old address-activation operation. Consequently, the address-activation operation may be re-executed by the second thread. However, since all our address-activation operations are idempotent, their re-execution does not produce further effect. Once again, idempotence is of critical importance.

Therefore, the issue of when to activate a target address has to be carefully studied considering the aforementioned constraints. πCFI selects different design points for different kinds of target addresses, including return addresses, function addresses, virtual method addresses, and addresses associated with exception handlers. Each kind of these target addresses has different activation sites, which will be discussed next. Without losing generality, we use x86-64 Linux to discuss the technical details. As we will see, activation of target addresses is the result of a careful collaboration between πCFI's compilation toolchain, its loader, and its runtime.

4.2.1 Return addresses

The most common kind of indirect-branch targets is return addresses. A return address could be activated immediately before a return instruction. However, it would not be an idempotent operation as the same return instruction may return to a different return address next time it is run. Instead, πCFI activates a return address when its preceding call instruction is executed. The activation procedure is different between direct calls and indirect calls, which are discussed separately next.

For a direct call, we use the example in Figure 4 to illustrate its activation procedure. To activate return address `L` following a direct call to `foo`, the following steps happen:

1. Before the direct call, πCFI's compilation toolchain inserts appropriate no-ops (line 3) to align `L` to an 8-byte aligned address. πCFI's implementation is based on MCFI [19], which requires all target addresses are 8-byte aligned.

2. When the code is loaded into memory by πCFI's loader (part of its runtime), the immediate operand of the call instruction (line 4) is replaced with an immediate called `patchstub`, as shown in Figure 4 (b). Therefore, the call is redirected to `patchstub`, whose code is listed in Figure 5.

3. When line 4 is reached after the program starts execution, the control transfers to `patchstub`. It firstly pops the return address `L` from the stack (line 2 in Figure 5) to `%r11`, which can be used as a scratch register thanks to the calling convention of x86-64 Linux. It then invokes `return_address_activate` provided by πCFI's runtime.[2]

4. The runtime, once entered, saves the context and activates `L` by updating the ECFG. πCFI reuses MCFI's tables for encoding an SCFG. There is a table called Tary in MCFI that lists all valid target addresses. So activating an address means an update to the Tary table to enable the address; we will discuss the detailed address-activation method through the Tary table in the implementation section.

5. The runtime then copies out eight bytes from [`L-8`, `L`), modifies the immediate operand of the call instruction to target `foo`, and uses an 8-byte move instruction to patch the code, as shown in Figure 4 (c). Finally, the runtime restores the context and jumps to line 4 in Figure 4 (c) to execute the patched call instruction.

```
1 // (a) before  // (b) after     // (c) after
2 //     loading //     loading   //     patching
3  nop            nop              nop
4  call foo       call patchstub   call foo
5 L:             L:               L:
```

Figure 4: How πCFI activates a return address following a direct call instruction. `L` is 8-byte aligned.

```
1 patchstub:
2   pop %r11
3   jmp %gs:return_address_activate
```

Figure 5: The patch stub for activating return addresses.

A few points are worth further discussion. First, since any return address is 8-byte aligned and any direct call instruction is 5-byte long, 8-byte atomic code update is always feasible and consequently all threads either call `patchstub` or `foo`. Second, the ECFG update should always be conducted prior to the update that changes `patchstub` to `foo`; otherwise another thread would be able to enter `foo`'s code and execute `foo`'s return instruction without `L` being activated.

Finally, the `patchstub` uses the stack to pass the address to be activated and therefore there is a small time window between the call to `patchstub` and the stack-pop instruction in `patchstub` during which an attacker can modify the return address on the stack. However, the most an attacker can do is to activate a different valid target address because the πCFI runtime would reject any invalid target address according to the SCFG. More importantly, since there are CFI checks before return instructions, CFI will never get violated. If we want to guarantee that πCFI always activates the intended address, one simple way would be to load the return address to a scratch register and pass the value to `patchstub` via the scratch register. This would add extra address loading instructions and no-ops after patching. Another way would be to have a dedicated patch stub for each call instruction (instead of sharing a patch stub among all call instructions and relying on the stack for passing the return address). This solution would cause roughly the same runtime overhead, at the cost of additional code bloat (around 14% on average for SPECCPU2006 C/C++ benchmarks).

Next we describe how πCFI activates return addresses following indirect calls. Only indirect calls through registers are emitted in πCFI-compiled code, as all indirect calls through memory are translated to indirect calls through registers. The instrumentation is listed in Figure 6. The `cfi-check` at line 3 is an operation that performs CFI checks and can be implemented using any conventional CFI (our implementation uses MCFI checks [19]). The `cfi-check` also contains no-ops to align `L` to an 8-byte aligned address. In addition, πCFI inserts a 5-byte no-op (line 4) at compile-time (Figure 6 (a)) so that at load time a direct call to the `patchstub` can be inserted (Figure 6 (b)). Note that in this case when `patchstub` gets called its stack pop instruction (line 2 in Figure 5) does not load `L` to `%r11`, but the runtime can straightforwardly calculate `L` by rounding `%r11` to the next 8-byte aligned address. After the return address is activated by the runtime, the `patchstub` call is patched back to the 5-byte no-op (Figure 6 (c)). The patch is atomic because an indirect call instruction through a

[2] The `%gs` segment register points to an area outside of the πCFI sandbox.

```
// (a) before  // (b) after      // (c) after
//      loading //      loading   //      patching
  cfi-check %r8  cfi-check %r8     cfi-check %r8
  nop // 5-byte  call patchstub    nop
  call *%r8      call *%r8         call *%r8
L:             L:                L:
```

Figure 6: How πCFI activates a return address following an indirect-call instruction. L is 8-byte aligned.

```
1 void foo(void) {}
2 void bar(void) {}
3 void (*fp) = &foo;
4 int main() {
5   void (*bp) = &bar;
6   fp();
7   bp();
8 }
```

Figure 7: Example code for function address activation.

register in x86-64 is encoded with either 2 or 3 bytes; therefore, the patched bytes will always stay within [L-8, L).

4.2.2 Function addresses

As discussed before, we cannot activate the target address immediately before an indirect call because of the idempotence requirement. Instead, πCFI activates the address of a function at the place when the function's address is taken. Consider an example shown in Figure 7, where foo and bar are global functions. foo's address is taken at line 3, while bar's address is taken at line 5. For those functions whose addresses are taken in the global scope, such as foo, πCFI activates their addresses at the beginning of execution; hence no additional instrumentation and patching are required for these function addresses. For functions whose addresses are taken elsewhere, such as bar, πCFI inserts address-activation operations right before their address-taking sites. As an example, Figure 8 presents part of the code that is compiled from the example in Figure 7 and the lea instruction at line 4 in Figure 8 takes the address of bar. Before the instruction, πCFI's compilation inserts a direct call to patchstub_at (at line 2 in Figure 8 (a)), which is another stub similar to Figure 5 but invokes a separate runtime function) to activate bar's address. However, a mechanism is required to translate the value passed on stack into bar's address, which is achieved by the label ("__picfi_bar") inserted at line 3. The label consists of a special prefix ("__picfi_") and the function's name (bar), so the runtime can look up the symbol table to translate the stack-passed value to the function's name during execution, and then looks up the symbol table again to find the address of bar. Appropriate no-ops are also inserted before line 2 so that the 5-byte patchstub_at call instruction ends at an 8-byte aligned address to enable atomic patching. The patching replaces the call instruction with a 5-byte no-op shown in Figure 8 (b).

C++ code can also take the address of non-virtual methods. Such an address is activated in the same way as a function address; that is, it is activated at the place where the address is taken.

Besides, functions' addresses can also be explicitly taken at runtime by the libc function dlsym. Therefore, we changed dlsym's implementation so that before dlsym returns a valid function address, a πCFI runtime trampoline is called to activate the function's address.

```
1 // (a) after loading    // (b) after patching
2   call patchstub_at       nop // 5-byte
3 __picfi_bar:            __picfi_bar:
4   lea bar(%rip), %rcx     lea bar(%rip), %rcx
```

Figure 8: πCFI's instrumentation for activating a function address.

```
1  class A {
2    public:
3      A() {}
4      virtual void foo(void) {}
5      virtual void bar(void) {}
6  };
7  class B : A {
8    public:
9      B() : A() {}
10     virtual void foo(void) {}
11 };
12 int main() {
13   B *b = new B;
14   b->foo();
15   A *a = new A;
16   a->foo();
17 }
```

Figure 9: Example C++ code for demonstrating virtual methods' address activation.

4.2.3 C++ virtual method addresses

πCFI activates a virtual method's address when the first object of the virtual method's class is instantiated. Consider the code example in Figure 9. Methods A::bar and B::foo's addresses are activated at line 13, because class B has foo declared and inherits the bar method from class A. Method A::foo's address is activated at line 15.

In πCFI, the address-activation operations for virtual method addresses are actually inserted into the corresponding classes' constructors so that, when a constructor gets first executed, all virtual methods in its virtual table are activated. For example, suppose Figure 10 (a) shows the prologue of A's constructor A::A, which is 8-byte aligned. When the code is loaded into memory, as shown in Figure 10 (b), πCFI's runtime changes the prologue to a direct call to patchstub_vm (which is another stub similar to patchstub in Figure 5 but jumps to a separate runtime function to activate virtual methods) so that, when A::A is firstly entered, the virtual method activation is carried out. Note that in this case when patchstub_vm is executed, its stack pop instruction (same as line 2 in Figure 5) does not set %r11 as the constructor's address, so the runtime needs to calculate it by taking the length of the patchstub_vm call instruction (5 bytes) from %r11. After its first execution, the runtime patches the direct call back to its original bytes, and executes the actual code of A::A. Only five bytes are modified in the patching process, and all these five bytes reside in an 8-byte aligned slot; therefore, the patch can be performed atomically.

The above virtual method activation procedure assumes a class object is always created by calling one of the class's constructors. Although most classes have constructors, there are exceptions. For example, due to optimization, some constructors might be inlined. We could either disable the optimization or activate the addresses of the associated virtual methods at the beginning of the program execution. πCFI chooses the latter method for simplicity and performance.

```
// (a) before   // (b) after       // (c) after
//     loading  //     loading     //  patching
A::A:           A::A:              A::A:
 push %rbp       call patchstub_vm  push %rbp
 mov %rsp,%rbp   ... // omitted     mov %rsp,%rbp
```

Figure 10: How πCFI activates a virtual method by instrumenting and patching a C++ class constructor A::A, which is 8-byte aligned.

4.2.4 *Exception handler addresses*

Exception handlers include code that implements C++ catch clauses and code that is generated by the compiler to release resources during stack unwinding[3]. We consider an exception handler's address activated when the function where the exception handler resides gets executed for the first time. Therefore, same as how πCFI instruments and patches C++ constructors, πCFI also instruments those functions that have exception handlers when loading the code into memory and patches the code back to its original bytes when such functions are executed for the first time.

4.3 Compatibility issues

As a defense mechanism, πCFI transforms an application to insert CFI checks and code for address activation, and performs online patching. We next discuss how this process is made compatible with typical programming conventions, including dynamic linking, process forking, and signal handling.

Dynamic linking. The ability to load/unload libraries dynamically is essential to modern software and makes it possible to share commonly used libraries across applications. πCFI's implementation is based on MCFI, designed to support modularity features such as dynamic linking and JIT compilation. Whenever a new library is dynamically loaded, MCFI builds a new Static CFG (SCFG) based on the original application together with the new library; the new SCFG will be installed and used from that point on.

πCFI's design of using address activation is also compatible with dynamic linking, based on the following reasoning. When an address, say $addr$, is activated, all edges with $addr$ as the target in the SCFG are implicitly added to the Enforced CFG (ECFG). Now suppose a library is dynamically loaded. It triggers the building of a new SCFG, which may allow more edges to target $addr$, compared to the old SCFG. However, since $addr$ has already been activated, the current ECFG allows an indirect branch to target $addr$ through newly added edges. Therefore, address activation accommodates dynamic linking.

πCFI also supports dynamic library unloading. When a library is unloaded, all indirect branch targets inside the library's code are marked inactive. This prevents all threads from entering the library's code, since there should be no direct branches targeting the library[4]. However, there might be threads currently running or sleeping in the library's code. Hence, it is unsafe to harvest the library code pages at this moment; otherwise those pages could be refilled with newly loaded library code and the sleeping threads might resume and execute unintended instructions. To safely handle this situation, πCFI asynchronously waits until it observes that all threads have executed at least one system call or runtime trampoline call; we instrumented each syscall instruction in the libc to increment a per-thread counter when a syscall instruction is executed. Then the runtime can safely reclaim the memory allocated for the library.

Process forking. In Linux, the fork system call is used to spawn child processes. For example, the Nginx HTTP server forks child processes to handle user requests. During forking, all non-shared memory pages are copied from the parent process to the child process (typically using a copy-on-write mechanism for efficiency). As a result, the child process has its own copy of the SCFG/ECFG data structure. This is good for security, because the child and the parent processes can grow their ECFGs separately as each has its own private copy of the data structure.

However, there is an issue with respect to the code pages. Recall that, to achieve secure code patching, the actual code pages and the shadow code pages are mapped to the same physical pages (as shown in Figure 3). In Linux, this is achieved by using mmap with the MAP_SHARED argument. As a result, the actual code pages are considered shared and the fork system call would not make private copies of the code pages in the child process. Consequently, we would encounter the situation of having shared code pages and private CFG data structures between the parent and the child processes. This would create the following possibility: the parent would activate an indirect branch target address, update its private ECFG, and patch the code; the child would lose the opportunity to patch the code and update its private ECFG, since the address-activation instrumentation would have been patched by the parent; the child's subsequent normal execution would be falsely detected as CFI violation.

To solve this problem, πCFI intercepts the fork system call, and before it is executed πCFI copies the parallel-mapped code pages to privately allocated memory and unmaps those pages. Then fork is invoked, which copies the private code pages as well. The runtimes in both processes next restore the parallel mapping in their own address spaces using the saved code bytes. This solution allows the child process to have its private code pages and CFGs. The same solution also applies to those parallel-mapped read-only data pages (shown in Figure 3). It should be pointed out that this solution does not support fork calls issued in a multi-threaded process, because the unmapping would crash the program if other threads are running. However, to the best of our knowledge, multi-threaded processes rarely fork child processes due to potential thread synchronization problems. Another downside of this approach is that it disables code sharing among processes, which would increase physical memory consumption.

Signal Handling. On Linux, signal handlers are usually not invoked by any application code[5], so they do not return to the application code. Instead, signal handlers return to a code stub set up by the OS kernel, which invokes the sigreturn system call. πCFI provides new function attributes for developers to annotate signal handlers in the source code and inlines the code stub into each signal handler during code generation. Each signal handler is associated with a special type signature, therefore it will never become any indirect call target. Moreover, since the signal handler is sandboxed in the same way as regular application code, the signal handling stack for each thread should be in the sandbox. Therefore, after a new thread is created, the libc code allocates a memory region inside the sandbox and executes sigaltstack to switch the stack to the in-sandbox region, which is released when the thread exits. Although the signal stack is allocated in the sandbox, we

[3] Compilers in Linux implement exception handling following the Itanium C++ ABI.

[4] This is generally true for libraries, but not for JITted code, in which case we need to check the remaining code for this condition.

[5] If a signal handler is invoked by application code, we can change the code to duplicate the handler so that the copy is never invoked.

believe it is hard for attackers to mount reliable attacks, and the reasons will be explained in Sec. 6.

5. IMPLEMENTATION

The πCFI toolchain basically has two tools: an LLVM-based C/C++ compiler, which is built on top of MCFI's compiler for code instrumentation and generation of SCFG-related metadata; and a runtime that loads instrumented modules and monitors their execution.

The πCFI compiler is modified from Clang/LLVM-3.5, with a `diff` result of 4,778 lines of changes. In summary, the MCFI-specific changes to LLVM propagate metadata such as class hierarchies and type information for generating the SCFG. The metadata are inserted into the compiled ELF as new sections. The instrumentation for indirect branches follows MCFI. For better efficiency, we applied the sandboxing method of ISBoxing [11] to instrument indirect memory writes. In detail, the sandbox for running applications is within [0, 4GB) [6], and the πCFI compiler instruments each indirect memory write instruction by adding a `0x67` prefix, which is the 32-bit address-override prefix. The prefix forces the CPU to clear all upper 32 bits after computing the target address. The πCFI-specific changes to LLVM identify function address-taking instructions and insert calls to `patchstub_at` before these instructions (detailed in Sec. 4). In addition, each πCFI-protected application runs with instrumented libraries. Therefore, we also modified and instrumented standard C/C++ libraries, including the `musl` libc, `libc++`, `libc++abi`, and `libunwind`.

The πCFI runtime consists of 11,002 lines of C/assembly code The runtime is position-independent, and is injected to an application's ELF as its interpreter. When the application is launched, the Linux kernel loads and executes the runtime first. The runtime then loads the instrumented modules into the sandbox region, creates shadow regions, and patches the code appropriately (detailed before in Sec. 4). The SCFG is generated using the metadata in the code modules, but initially all target addresses in the SCFG are made inactive (this encodes an implicit empty ECFG). The details of how SCFG is encoded can be found in the MCFI paper. As a summary, the SCFG is encoded as two tables: the Bary table remembers the IDs of all indirect branches; the Tary table records the IDs of all indirect branch targets. All IDs are 8-byte long and stored at 8-byte aligned addresses to enable atomic updates. For each indirect branch, MCFI's instrumentation retrieves its ID from Bary, and the intended target ID from Tary, then compares whether the IDs are equal to detect CFI violation. In each ID, there are several *validity bits* at fixed positions and with special values; invalid IDs do not have those bit values at those positions. As a result, to mark a target address inactive, πCFI simply changes the values of those validity bits to wrong values in the target's relevant Tary ID. During an address-activation operation, the πCFI runtime atomically marks the address active (if it is a valid target address) by changing the values of the validity bits back and patches the address-activation operation to no-ops.

6. SECURITY ANALYSIS

πCFI can mitigate both *code injection* and *code reuse* attacks. For code-injection attacks, πCFI enforces DEP (Data Execution Prevention) at all times; its runtime enforces this by intercepting and checking all systems calls that may change memory protection, including `mmap`, `mprotect` and `munmap`. Therefore, code injection attacks are impossible for programs that do not generate code at runtime. For programs that generate code on-the-fly (i.e., JIT compilers), their JITted code manipulation is performed by the trusted runtime following the work of RockJIT. Attackers may still inject code into the JITted code, but the injected code never violates CFI due to online code verification. For example, the injected code should never contain any system call instruction. Further, JIT spraying [5] attacks are also prevented, because CFI never allows indirect branches to target the middle of instructions.

For code-reuse attacks (e.g., ROP), πCFI mitigates them by enforcing a fine-grained per-input CFG, which provides multiple security benefits. First of all, when there is dead code in a program, its ECFG makes dead code unreachable. For instance, if a libc function is never called in an application's source code (including libraries), then it is unreachable at runtime. This property makes remote exploits nearly impossible for programs that never invoke critical system functions (e.g., `execve`) as most attacks rely on invoking such functions to cause damage. Examples of such programs include compression tools (e.g., gzip), bind (a widely-used DNS server), and memcached etc.; they do not invoke `execve`-like functions.

Second, by reducing indirect branch targets/edges, πCFI makes it hard for attackers to redirect the control flow from the first instruction they can control (e.g., an indirect branch) to their targeted sensitive function. Using MCFI's CFG as the baseline, we next present experiments that measure πCFI's indirect branch target/edge reduction.

On a machine running x86-64 Linux, we compiled and instrumented all 19 SPECCPU2006 C/C++ benchmark programs with the O3 optimization. We then measured the statistics of the enforced CFGs using the reference data sets that are included in the benchmarks. If a benchmark program has multiple reference data sets, we chose the one that triggered the most address-activation operations (i.e., the worst case). The results are shown in Table 1. The "RAA" column shows the percentage of return addresses that are activated at the end of the program over the return addresses in MCFI's CFG; the "FAA" column shows the percentage of activated function addresses over function addresses in MCFI's CFG (note that not all functions are indirect-branch targets in MCFI's CFG; if a function's address is never taken, then MCFI does not allow the function to be called via an indirect branch); the "VMA" column shows the percentage of activated virtual method addresses; the "EHA" column shows the percentage of activated exception handlers. Finally, "IBTA" column shows the percentage of all activated indirect branch target addresses, and the "IBEA" column shows the percentage of indirect-branch edges in πCFI's ECFG at the end of the program over the indirect-branch edges in MCFI's CFG. Those C programs (i.e., those above 444.namd in the table) do not have virtual methods or exception handlers; therefore, VMA and EHA measurements are not applicable to them.

As can be seen in the table, only a small percentage (10.4% on average) of indirect branch edges are activated in the ECFG. Most programs activate less than 20% of indirect branch edges, which severely limits attackers' capability of redirecting control flow. The low percentage of edge activation is mostly attributed to the low percentage of return address activation as return addresses are the most common kind of indirect-branch targets. Function addresses are activated in higher percentages. The reason is that C programs tend to take addresses of functions early in the program and store them in function-pointer tables. From the perspective of security engineering, it would be better to refactor such programs to dynamically take function addresses, following the principle of least privilege. In addition, to simulate real attack scenarios when attackers

[6] The maximum sandbox size can be extended to 64TB on x86-64 if the sandboxing technique in PittSFIeld [16] is used or the πCFI runtime is implemented as a kernel module.

Benchmark	RAA	FAA	VMA	EHA	IBTA	IBEA
400.perlbench	19.9%	83.2%	N/A	N/A	22.5%	15.4%
401.bzip2	5.0%	41.9%	N/A	N/A	5.6%	6.1%
403.gcc	27.0%	91.7%	N/A	N/A	28.6%	20.3%
429.mcf	5.5%	45.0%	N/A	N/A	6.1%	7.4%
433.milc	13.6%	41.9%	N/A	N/A	13.9%	9.6%
445.gobmk	35.4%	98.1%	N/A	N/A	43.4%	64.4%
456.hmmer	9.2%	32.9%	N/A	N/A	9.4%	9.4%
458.sjeng	9.8%	46.3%	N/A	N/A	10.3%	8.3%
462.libquantum	7.2%	39.3%	N/A	N/A	7.7%	8.3%
464.h264ref	19.5%	49.5%	N/A	N/A	20.0%	20.6%
470.lbm	4.5%	40.0%	N/A	N/A	5.1%	7.4%
482.sphinx	18.9%	44.8%	N/A	N/A	19.1%	14.8%
444.namd	5.3%	84.3%	61.5%	3.2%	8.9%	3.5%
447.dealII	7.1%	95.5%	32.2%	13.0%	10.7%	5.5%
450.soplex	8.9%	87.7%	69.8%	19.5%	14.2%	7.6%
453.povray	12.9%	92.1%	62.9%	5.3%	16.1%	9.6%
471.omnetpp	19.1%	94.8%	55.4%	37.7%	25.3%	13.9%
473.astar	5.3%	87.4%	61.2%	2.2%	8.9%	6.4%
483.xalancbmk	14.3%	94.5%	56.6%	27.9%	21.4%	13.5%

RAA: Return Address Activation; FAA: Function Address Activation; VMA: Virtual Method Activation; EHA: Exception Handler Activation; IBTA: Indirect Branch Target Activation; IBEA: Indirect Branch Edge Activation.

Table 1: ECFG statistics of SPECCPU2006 C/C++ programs.

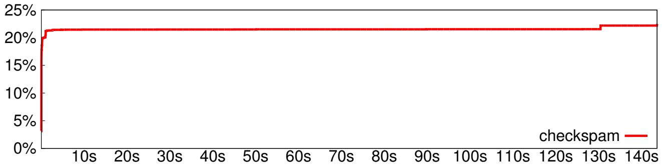

Figure 11: Growth of activated target addresses for 400.perlbench.

can feed multiple different inputs to a given program to trigger as many indirect branch targets as possible, we calculated the cumulative total indirect branch targets for 400.perlbench and 403.gcc by merging the activated addresses of each input file in both the test and reference data sets. For 400.perlbench, about 31.9% of indirect branch targets are cumulatively activated; for 403.gcc, around 34.9%. These numbers indicate that it might be hard to activate all indirect branches even with multiple inputs.

In our experiments, we were also interested in learning how the ECFG grows over time. For each benchmark, we measured the number of activated indirect branch targets over time. For most benchmarks (18 out of 19), most address activation happens at the beginning of execution and grows slowly (and stabilizes in most cases). For example, Figure 11 shows the target activation of the 400.perlbench program when tested on its longest-running data set `checkspam`. The X-axis is the execution time and the Y-axis is the proportion of activated indirect branch targets. However, we also observed an outlier, 403.gcc when tested over the `g23` data set, whose address activation curve is drawn in Figure 12. As can be seen, the address activation shows steep growth even at the end; on the other hand, it does not activate more target addresses compared to other input data sets, which trigger similar ECFG growth as 400.perlbench.

We also built and instrumented the Google V8 JavaScript engine as a standalone executable. We ran V8 on three benchmark suites: Sunspider 1.0.2, Kraken 1.1, and Octane 2. We collected ECFG statistics for those benchmark suites in Table 2. The first "No input" row shows the statistics when no input is fed to V8. Note that the benchmarks, especially Octane 2 (around 373K lines of JavaScript code) does not activate significantly more targets than the no-input case. When we merge all benchmarks' results, about 30% of indirect branch targets are activated in total, slightly more than the result triggered by Octane 2. Therefore, given the size and diversity of benchmarks, we hypothesize that other JavaScript programs will not activate significantly more addresses than those benchmarks. The ECFG growth curve of V8 when tested over Octane 2 is shown in Figure 13, from which we can see that the number of target activation grows very slowly after the initial burst, similar to what we observed on SPEC benchmarks.

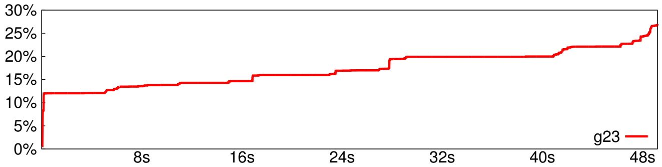

Figure 12: Growth of activated target addresses for 403.gcc.

Benchmark	RAA	FAA	VMA	EHA	IBTA	IBEA
No input	15.6%	86.5%	41.4%	2.2%	18.5%	17.8%
Sunspider 1.0.2	23.1%	86.8%	56.2%	2.2%	26.1%	24.9%
Kraken 1.1	21.8%	86.9%	53.9%	2.2%	24.8%	23.2%
Octane 2	26.6%	87.0%	59.2%	2.2%	29.5%	28.6%

Table 2: ECFG statistics of the Google V8 JavaScript engine.

One security concern about πCFI is that a long running program that keeps taking user input may be able to trigger the activation of all target addresses. For evaluation, we used πCFI to protect an Nginx server and used the sever to host a WordPress site. Then one of the authors used almost all features of WordPress for a session of about 20 minutes. Table 3 shows the address activation results. We configured Nginx to use two processes: the master process was responsible for initialization and handling administrators' commands while a worker process created by the master processed all user inputs. πCFI's design allows the master and worker to have different ECFGs; therefore their address activation results are different. Figure 14 shows the target activation growth curve for the worker process. Similar to other tested programs, the percentage quickly stabilized. This preliminary experiment shows that πCFI provides security benefits even for long running programs. As discussed before, we believe this is because programs have a large amount of unused code for a particular input, including exception-handling code, library code, and code for handling different configurations.

Next we briefly discuss how recently proposed attacks are mitigated by πCFI.

COOP attacks. Counterfeit Object-Oriented Programming (COOP [25]) attacks construct counterfeit C++ objects with forged virtual table pointers and carefully chain virtual calls to achieve Turing-complete computation, even in applications protected with coarse-grained CFI. However, as mentioned by the authors of COOP, CFI solutions that generate fine-grained CFGs based on class hierarchies tend to be immune to COOP. Since πCFI builds static CFGs using class hierarchies (similar to SafeDispatch) and performs online activation of virtual methods, COOP is made harder on πCFI-protected C++ applications.

CFB attacks. Control-Flow Bending (CFB, [7]) is a recently proposed general methodology for attacking conventional CFI systems that statically generate CFGs. At a high level, CFB abuses certain functions (called dispatchers) whose execution may change their own return addresses to "bend" the control flow. These dispatchers are often common libc routines (e.g., `memcpy`, `printf`, etc.) that

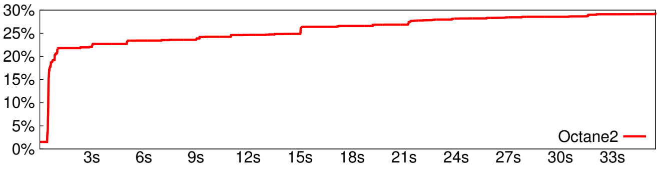

Figure 13: Growth of activated target addresses for V8.

Benchmark	RAA	FAA	IBTA	IBEA
Master	9.3%	67.1%	13.3%	8.6%
Worker	14.9%	73.5%	19.0%	13.2%

Table 3: ECFG statistics of the Nginx HTTP server's master and worker processes.

are invoked in many places of the program to increase the possibility of bending the control flow to critical system call sites. Different dispatchers can be chained to achieve more flexibility. For example, the authors show that `memcpy` and `ngx_snprintf` in Nginx can be used as dispatchers to alter the normal control flow to another site where `execve` is reachable.

πCFI mitigates CFB attacks by reducing the number of return addresses of dispatchers. For example, the same exploit to attack Nginx in the CFB paper would fail in πCFI-protected Nginx, because the dispatcher chain will be cut off due to inactive return addresses in the worker process. (Specifically, `ngx_exec_new_binary` is not executed in the worker process, so all return addresses inside it won't be activated.) The xpdf exploit in the CFB paper can also be prevented since it does not use `execve`-like functions, which makes those functions unreachable. To attack πCFI, attackers may firstly need to steer the control flow to activate return addresses of their interest.

On top of πCFI, we can further mitigate CFB attacks by disabling certain dispatchers. For example, in our implementation, `memcpy` is changed so that its return address is stored in a dedicated register once it is entered. When `memcpy` returns, the value stored in the register is used. Similar changes are also made to other `strcat`-like libc functions. With these changes, the attackers can no longer directly use those functions as dispatchers, although they can still use the callers of those functions. As long as those function's callers do not have as many call sites as those functions, what the attackers can do becomes more restricted.

SROP attacks. SROP [6] attacks exploit the Linux signal handling mechanism to mount attacks. Specifically, when a signal is delivered to a thread, the Linux kernel suspends the thread and saves its context (e.g., program counter and stack pointer) onto the thread's signal handling stack in user space and invokes the signal handler. After the signal handler finishes execution, it returns to a kernel-inserted stub calling `sigreturn`, which restores the saved thread context. Therefore, it is possible that attackers may change the saved context or fake one and redirect the control flow to a `sigreturn` system call and restore the context to execute arbitrary code. πCFI mitigates SROP attacks by inlining `sigreturn` system calls into each signal handler, which is unreachable from other application code. As a result, attackers need to trigger real signals to execute the `sigreturn` system call. To corrupt the saved thread context, the attackers have to either exploit a buggy signal handler, or use other threads to concurrently and reliably modify the signal handling thread's saved context, neither of which we believe is easy since signal handlers rarely have complex code and usually do not run for an extended period of time.

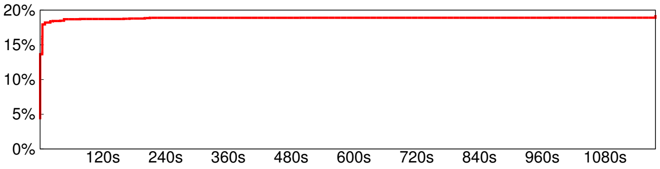

Figure 14: Growth of activated target addresses for Nginx.

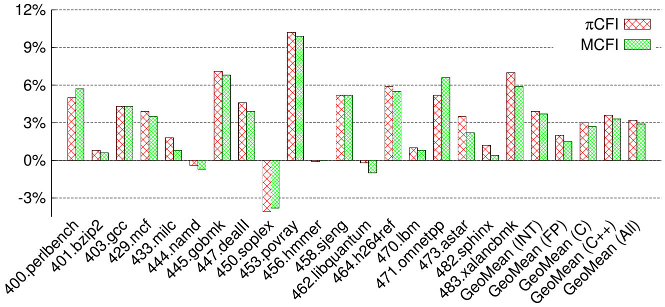

Figure 15: πCFI and MCFI runtime overhead on SPECCPU2006 C/C++ benchmarks.

7. PERFORMANCE EVALUATION

As a security mechanism, πCFI's performance overhead should be small to have a chance of being adopted in practice. πCFI's design is geared toward having a small runtime overhead, including the use of idempotent operations and online code patching. Next we report our experiment results of the performance overhead of πCFI, including runtime and space overhead. Of the two, having a small runtime overhead is much more important. All performance numbers were measured on a system with an Intel Xeon E3-1245 v3 processor, 16GB memory, and 64-bit Ubuntu 14.04.2. For comparison, we also measured MCFI's performance overhead using the same machine and compilation configuration.

SPECCPU2006. The SPECCPU2006 benchmark suite consists of 19 C/C++ benchmarks. We compiled all of them at the O3 optimization level using the πCFI compilation toolchain and measured their execution time increase compared to their counterparts that were compiled with a native Clang/LLVM compiler. Each benchmark was executed three times over its own reference data sets with a less than 1% standard deviation. The runtime-overhead results are presented in Figure 15. On average, πCFI incurs 3.9% overhead on integer benchmarks and 3.2% overhead over all benchmarks (including both integer and floating-point benchmarks). In comparison, MCFI incurs 3.7% and 2.9% on the same benchmark sets. We note that the original MCFI paper [19] reports around 5% runtime overhead. The major difference is because we changed its way of sandboxing indirect memory writes to use a more efficient technique (following [11]). Compared to MCFI, πCFI causes a small increase of runtime overhead, due to address-activation operations and execution of no-ops after patching.

The figure also shows that there are a few benchmarks (e.g., 450.soplex) that have slight speedups after πCFI's protection. To investigate the phenomenon, we replaced the instrumentation with no-ops and still observed speedups; so they should be resulted from extra alignments added during compilation (the same speedup phenomenon has been observed by other work in the SFI/CFI literature). Another note is that the runtime overhead is positively correlated with the frequency of indirect branches executed in a run. We measured the correlation using the Pearson correlation coefficient and got the result of 0.74, which indicates strong correlation.

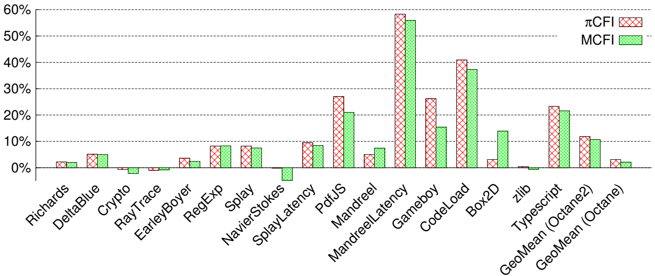

Figure 16: πCFI and MCFI runtime overhead on Octane 2 benchmarks with Google V8.

Due to instrumentation, the code size for benchmarks is enlarged by 5.6% to 67.4%, with an average of 21.2%. Code size of C++ programs increases more than C programs, due to a higher density of indirect branches.

V8. Following the RockJIT approach [20], we instrumented the V8 JavaScript engine (version 3.29.88.19 on x86-64) using πCFI's compiler with the release mode, and measured the performance overhead on the Octane 2 benchmarks. Figure 16 shows the runtime-overhead results. On average, 11.8% runtime overhead is incurred by πCFI, while 10.7% by MCFI. As analyzed before, πCFI still costs a bit more time than MCFI due to online address activation and patched no-ops. Also, we separately calculated runtime-overhead results for the subset of benchmarks that were included in Octane 1 (the predecessor of Octane 2) since past work used Octane 1 for evaluation. πCFI incurs only 3.1% overhead over them on average, which is slightly higher than MCFI (or RockJIT). Compared to other JIT-compiler hardening work, such as NaCl-JIT [3], librando [14], and SDCG [28], πCFI incurs less overhead and provides better security. In terms of code bloat, the code size of V8 increases by around 35.7% after the πCFI instrumentation.

Nginx. We compiled nginx-1.4.0 with the O2 optimization and measured its throughput. Using the ab command, we found that the binary hardened by πCFI had about the same maximum throughput as the native version. The code size increases by about 22.3%.

8. FUTURE WORK

In πCFI, the ECFG grows monotonically if no code is unloaded. A larger ECFG decreases the strength of the CFI protection. As a result, a potential future direction is to study when to *deactivate addresses* safely. In general, an address of an application can be deactivated at a specific moment if no future execution of the application's code will reach that address. This notion is very similar to garbage data as defined in a garbage collector, except it is for code instead of data. Therefore, one idea is to design specialized garbage collectors for code to automatically compute what code is garbage and use that information to deactivate addresses in garbage code. Another way of address deactivation is to expose APIs to applications to deactivate addresses and ask developers to decide when and where to invoke these APIs. This is in general unsafe (similar to manual memory management in C/C++). However, it is still useful in situations when developers know exactly what code is inactive at a specific point.

We would also like to investigate how πCFI can be implemented in other architectures. Its design relies on the following hardware-provided mechanisms: (1) virtual memory, based on which the secure code patching is implemented; (2) atomic instruction-stream modification, which prevents hardware threads from executing corrupted instructions during patching. x86-32 and x86-64 CPUs support both, but it would be interesting to explore other CPU architectures such as ARM and MIPS.

9. CONCLUSIONS

We have presented πCFI, a new CFI technique that is able to generate and enforce per-input CFGs. πCFI builds on top of conventional CFI (specifically, MCFI in our implementation) with extra instrumentation inserted to the target program. It takes advantage of MCFI's CFG generation to compute a fine-grained static CFG for the program. During execution, πCFI dynamically activates target addresses lazily before the addresses are needed by later execution. Our evaluation shows that πCFI can effectively reduce available indirect branch edges by a large percentage, while incurring low overhead.

10. ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. We also thank Michael Spear and Stephen McCamant for useful discussions. This research is supported by US NSF grants CNS-1408826, CCF-1217710, CCF-1149211, and a research award from Google.

11. REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-Flow Integrity. In *12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[2] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. Preventing Memory Error Exploits with WIT. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (May 2008), pp. 263–277.

[3] Ansel, J., Marchenko, P., Erlingsson, Ú., Taylor, E., Chen, B., Schuff, D., Sehr, D., Biffle, C., and Yee, B. Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 355–366.

[4] Arias, O., Davi, L., Hanreich, M., Jin, Y., Koeberl, P., Paul, D., Sadeghi, A.-R., and Sullivan, D. HAFIX: Hardware-Assisted Flow Integrity Extension. In *52nd Design Automation Conference (DAC)* (June 2015).

[5] Blazakis, D. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (2010), WOOT'10, USENIX Association, pp. 1–9.

[6] Bosman, E., and Bos, H. Framing signals - a return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on* (May 2014), pp. 243–258.

[7] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association.

[8] Carlini, N., and Wagner, D. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[9] Davi, L., Dmitrienko, R., Egele, M., Fischer, T., Holz, T., Hund, R., Nurnberger, S., and Sadeghi, A. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[10] DAVI, L., LEHMANN, D., SADEGHI, A., AND MONROSE, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[11] DENG, L., ZENG, Q., AND LIU, Y. ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In *ICT Systems Security and Privacy Protection*, H. Federrath and D. Gollmann, Eds., vol. 455 of *IFIP Advances in Information and Communication Technology*. Springer International Publishing, 2015, pp. 386–400.

[12] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[13] GOKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out Of Control: Overcoming Control-Flow Integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (May 2014).

[14] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Librando: Transparent Code Randomization for Just-In-Time Compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security* (2013), CCS '13, ACM, pp. 993–1004.

[15] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 147–163.

[16] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (2006), USENIX-SS'06, USENIX Association.

[17] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)* (San Diego, California, February 2015).

[18] NIU, B., AND TAN, G. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), CCS '13, ACM, pp. 199–210.

[19] NIU, B., AND TAN, G. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 577–587.

[20] NIU, B., AND TAN, G. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1317–1328.

[21] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *22nd Usenix Security Symposium* (2013).

[22] PAYER, M., BARRESI, A., AND GROSS., T. R. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (Milano, Italy, July 2015).

[23] PEWNY, J., AND HOLZ, T. Control-Flow Restrictor: Compiler-based CFI for iOS. In *ACSAC '13: Proceedings of the 2013 Annual Computer Security Applications Conference* (2013).

[24] SAHA, S., LOZI, J.-P., THOMAS, G., LAWALL, J., AND MULLER, G. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems . In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* (June 2013), pp. 1–12.

[25] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).

[26] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th Usenix Security Symposium* (2010), pp. 1–12.

[27] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 552–561.

[28] SONG, C., ZHANG, C., WANG, T., LEE, W., AND MELSKI, D. Exploiting and Protecting Dynamic Code Generation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014* (2015).

[29] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[30] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), SOSP '93, ACM, pp. 203–216.

[31] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on* (May 2010), pp. 380–395.

[32] YEE, B., SEHR, D., DARDYK, G., CHEN, J., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Security and Privacy, 2009 IEEE Symposium on* (May 2009), pp. 79–93.

[33] ZENG, B., TAN, G., AND MORRISETT, G. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 29–40.

[34] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 559–573.

[35] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), SEC'13.