2016

Practical Control-Flow Integrity

Ben Niu
Lehigh University

Practical Control-Flow Integrity

by

Ben Niu

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy in

Computer Science

Lehigh University

January, 2016

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Ben Niu

Practical Control-Flow Integrity

Date

Gang Tan, Dissertation Director

Accepted Date

Committee Members

Gang Tan (Chair)

Mooi-Choo Chuah

Michael Spear

Stephen McCamant

ACKNOWLEDGEMENTS

I am very grateful to my great advisor Prof. Gang Tan. I had no idea how to do research when I came to Lehigh, and he taught me what to think of, how to think, how to present my thoughts, and how to write them down. His door was always open for discussion, he replied emails during weekends, and he provided detailed guidance on any technical problem. I was constantly surprised at his breadth of knowledge and depth of understanding, and I enjoyed working with him.

I also thank my thesis committee members, Prof. Mooi-Choo Chuah, Prof. Michael Spear, and Prof. Stephen McCamant for their courses, research works and insightful comments on my research. Michael's parallel programming course introduced me lock-free algorithms and his operating system course consolidated my system programming skills. Stephen's research on SFI provided me a foundation in my research, without which I would be still looking for decent solutions.

I appreciate my colleagues at MSRC during my internship at Microsoft, especially Matt & Ken, Suha, Vishal, Swamy, Joe, Axel and Marcus. Discussion with them about the design of Control-Flow Guard helps me better understand practicality of CFI.

I owe a big debt of gratitude to my apprehensive wife Yinzi, without whose company I could never finish this dissertation. I also thank her for bringing our daughter Jill to the world, who brought our family hope and joy, and I could not imagine how my life would be without her. Moreover, I am very thankful for the invaluable help and support my parents and parents-in-law provided during this adventure.

Last but not least, I thank my friends Zheng Shi, Shen Liu, Qi Li, Junqi Chen, Zhongyuan Wei, Zhongliang Liang, Mengtao Sun, Dongrui Zeng, Zi Wang, Bin Zeng and Siliang Li for their generous help in my daily life.

To my family.

Contents

List of Tables

List of Figures

ABSTRACT

Control-Flow Integrity (CFI) is effective at defending against prevalent control-flow hijacking attacks. CFI extracts a control-flow graph (CFG) for a given program and instruments the program to respect the CFG. Specifically, checks are inserted before indirect branch instructions. Before these instructions are executed during runtime, the checks consult the CFG to ensure that the indirect branch is allowed to reach the intended target. Hence, any sort of control-flow hijacking would be prevented.

However, CFI traditionally suffered from several problems that thwarted its practicality. The first problem is about precise CFG generation. CFI's security squarely relies on the CFG, therefore the more precise the CFG is, the more security CFI improves, but precise CFG generation was considered hard. The second problem is modularity, or support for dynamic linking. When two CFI modules are linked together dynamically, their CFGs also need to be merged. However, the merge process has to be thread-safe to avoid concurrency issues. The third problem is efficiency. CFI instrumentation adds extra instructions to programs, so it is critical to minimize the performance impact of the CFI checks. Fourth, interoperability is required for CFI solutions to enable gradual adoption in practice, which means that CFI-instrumented modules can be linked with uninstrumented modules without breaking the program.

In this dissertation, we propose several practical solutions to the above problems. To generate a precise CFG, we compile the program being protected using a modified compilation toolchain, which can propagate source-level information such as type information to the binary level. At runtime, such information is gathered to generate a relatively precise CFG. On top of this CFG, we further instrument the code so that only if a function's address is dynamically taken can it be reachable. This approach results in lazily computed per-input CFGs, which provide better precision. To address modularity, we design a lightweight Software Transactional Memory (STM) algorithm to synchronize accesses to the CFG's data structure at runtime. To minimize the performance overhead, we optimize the CFG representation and access operations so that no heavy bus-locking instructions are needed. For interoperability, we consider addresses in uninstrumented modules as special targets and make the CFI instrumentation aware of them. Finally, we propose a new architecture for Just-In-Time compilers to adopt our proposed CFI schemes.

Chapter 1

Introduction

In this chapter, we first introduce prevalent control-flow hijacking attacks and deployed mitigation. Then, we describe concepts and implementations of Control-Flow Integrity (CFI, [1]), which is a fundamental approach to mitigating control-flow hijacking attacks. Finally, we present practical issues of previous CFI systems and summarize how we address those problems.

1.1 Control-Flow Hijacking

C and C++ are perhaps the most important programming languages, since almost all critical software is written in them. For instance, operating system kernels such as Windows and Linux, virtual machines such as Xen and JVM, compilers such as GCC and LLVM, database management systems such as MySQL and PostgreSQL, web browsers such as Chrome and FireFox, are all written mostly in C/C++. On the one hand, programs written in C/C++ enjoy high performance; on the other hand, these programs all suffer from memory corruption issues such as out-of-bound accesses or object use-after-free bugs.

Consider a real memory corruption bug found in the popular Nginx HTTP server 1.4.0, which is extracted and shown in Figure 1.1. An attacker has full control over `content_length_n` at line 8, which is a signed integer. The macro `ngx_min` at line 7 processes two signed integers and returns the least one. Hence, if the attacker feeds Nginx a negative value, `ngx_min` will return the negative

```
 1 #define ngx_min(val1, val2)  ((val1 > val2) ? (val2) : (val1))
 2 #define NGX_HTTP_DISCARD_BUFFER_SIZE       4096
 3 ...
 4 u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
 5 ...
 6 /* content_length_n is of type off_t, a signed integer type */
 7 size = (size_t) ngx_min(
 8   r->headers_in.content_length_n, /* attacker-controlled */
 9   NGX_HTTP_DISCARD_BUFFER_SIZE);
10
11 n = r->connection->recv(r->connection, buffer, size);
```

Figure 1.1: A stack buffer overflow bug (CVE-2013-2028) in Nginx 1.4.0.

integer, which will then be converted to an unsigned integer and assigned to `size` at line 7. Later on, the code invokes the `recv` system call at line 11 to populate the array `buffer` defined at line 4 with attacker-controlled data. Since the array length is smaller than the `size` returned at line 7, the array will overflow, which results in code injection or code reuse attacks.

Code injection. In code injection attacks, the attacker injects new code into the address space of the victim program and executes her code. In the example of Figure 1.1, if the stack is executable, the attacker could send malicious code bytes to fill the array as well as the overflown stack. During the same process, the return address of the current function can also be overwritten with a pointer pointing to the entry of the injected code. Then, when the current function returns, instead of returning to the caller, the function returns to the injected code and executes it.

Code reuse. In code reuse attacks, the attacker reprograms the existing code bytes to execute a malicious instruction stream. In the above example, the attacker could simply overwrite the return address to the address of a libc function `system` and write arguments to the function on the stack. When the function returns, `system` will be executed with attacker-fed arguments, which enable the attacker to execute arbitrary commands (subject to Nginx's security privileges) on the victim machine. As can be seen, no new code bytes are executed in code reuse attacks, but existing code is reused.

In cases where simply overwriting a return address is not sufficient to mount an attack (e.g., the arguments to sensitive functions cannot be arbitrarily set), attackers can use a more advanced code reuse technique called Return-Oriented Programming (ROP [2]). To mount such an attack, the attackers first scan the code bytes and find gadgets, which are instruction sequences ending with a return (or an indirect call/jump), and perform basic operations such as an addition or memory load. Then, by carefully overflowing the stack, the attackers can chain these gadgets into an arbitrary program.

Next, we use a simple proof-of-concept example in Figure 1.2 to demonstrate principles of ROP attacks in x86-64 Linux. We assume a function `foo` contains an out-of-bound array write bug that allows the attackers to overwrite the stack buffer, like the example in Figure 1.1. We also assume a piece of code at memory address `addr`, which normally performs arithmetic operations. However, if we decode the instruction stream from the middle, we could possibly get three ROP gadgets. Gadget_1 increments register `%eax` by one and returns; Gadget_2 sets `%eax` to zero and returns; and Gadget_3 performs a system call. As a result, if attackers overwrite `foo`'s return address to the address of Gadget_2, and write 231 copies of Gadget_1 addresses, followed by the address of Gadget_3, as shown on the right in Figure 1.2, the attackers can essentially set `%eax` to 0, increment it by 231 (equivalent to setting `%eax` to 231), and execute a system call after `foo` returns. Since Linux uses `%eax` to pass the system call number to the kernel, and 231 is the system call number for `exit_group`, the current process will exit. Although this simple attack only terminates the victim program, by carefully choosing and chaining gadgets, the attackers can conduct arbitrary computation on the victim system.

In both code injection and code reuse attacks, execution of the attack code almost always deviates from the legal control flow of the victim program. In other words, the attackers typically mount *control-flow hijacking* attacks by exploiting memory corruption bugs.

1.2 Deployed Defenses

It is extremely hard to find and fix all the memory corruption bugs before software is shipped to end users. Therefore, modern OSes (e.g., Windows, Linux and OSX) deploy the following bug-tolerant defenses to mitigate control-flow hijacking attacks.

Data-Execution Prevention (DEP) DEP prevents code injection attacks by enforcing that the stack

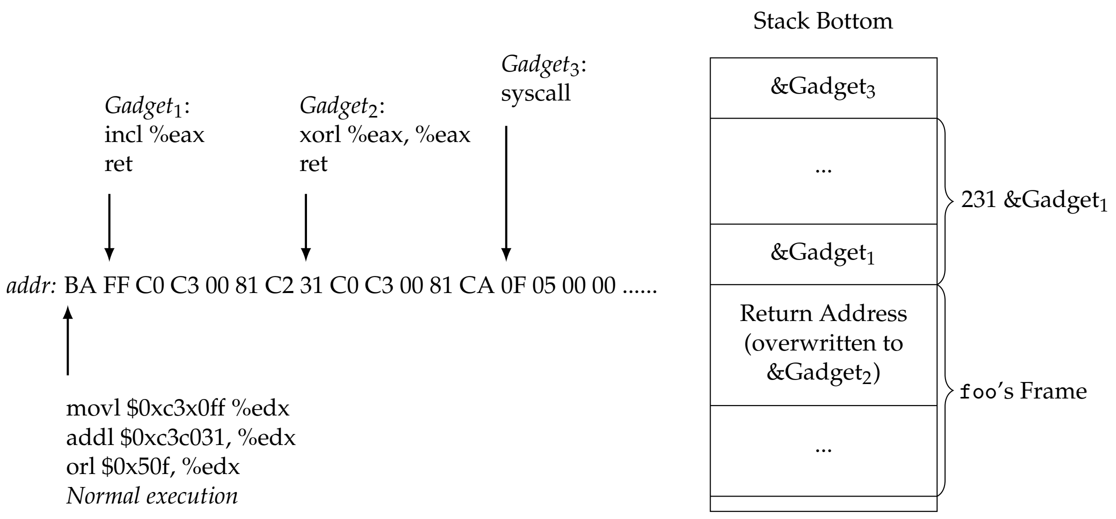


Figure 1.2: A proof of concept example of Return-Oriented Programming (ROP) attacks. Function `foo` contains a bug that enables stack buffer overflow. "&Gadget$_n$" means the address of Gadget$_n$. For example, &Gadget$_1$ = `addr+1`. Function `foo`'s stack frame is omitted except its return address.

and heap areas of a program are non-executable, while the code pages are non-writable by default. To enjoy DEP, current programs are compiled with separated code and data, which are stored in memory pages with different protection. However, it is easy to see that DEP has no protection against code reuse attacks. Moreover, DEP is not compatible with programs that generate and modify code on-the-fly, such as Just-In-Time (JIT) compilers.

Address Space Layout Randomization (ASLR) ASLR is an approach to mitigating both code injection and reuse attacks. The basic idea of ASLR is to make it harder for attackers to precisely locate the injected code or reusable code. To do so, program modules such as the executable file and dependent libraries are compiled to be position-independent and loaded to random places at runtime. Similarly, stack and heap are allocated at random memory addresses as well. Consequently, the attackers probably need to guess the position of code and data to mount a successful attack.

However, ASLR has its weaknesses. First, its security depends on how much entropy it can provide for randomization. As shown in [3], even brute-force guesses can be used to practically break low-entropy ASLR especially on 32-bit machines.

Second, ASLR is vulnerable to information leaks. For example, if the attacker can read a C++ object on the heap, she can easily figure out where the code modules are by following the virtual table pointer.

Third, ASLR can be degraded by heap spraying, whose basic idea is to allocate a large chunk of heap memory and fill that memory with repeated nops followed by the actual exploit code. Then, instead of precisely jumping to the exploit, the attacker could just randomly jump to any address in the chunk. Since most of the chunk is filled with nops, which will lead to execution of the exploit, the chance of success is high.

Stack Cookies Stack cookies [4] also mitigate control-flow hijacking attacks by monitoring the integrity of return addresses. When a function is called, its prologue pushes a random value (cookie) onto the stack and the epilogue checks whether the cookie has been modified. If the cookie changes, stack overflow might have happened and the program is terminated.

However, stack cookie can be bypassed. First, stack cookie only detects sequential stack buffer overflow. Unfortunately, if an attacker can control the index to an array, she can choose an index that bypasses the cookie. In addition, stack cookies cannot detect stack buffer underflow. Second, since the cookie is a randomly chosen value, it may be quickly guessed, as shown in [5]. Third, stack cookies cannot protect heap buffer overflows.

1.3 Control-Flow Integrity

Stronger than all the deployed defenses, Control-Flow Integrity (CFI [1]) is a fundamental method for mitigating control-flow hijacking attacks by forbidding illegal control flows. In general, CFI specifies a Control-Flow Graph (CFG) $G = (V, E)$ for a victim program P. In G, vertices V represent instructions and edges E denote legal control-flow transfers between instructions. By enforcing control-flow integrity with respect to G, it is guaranteed that every control-flow transfer during P's execution is in E.

Control-flow transfers can be either direct or indirect. Direct edges include sequential instruction execution and direct branching. For example, the transfer from a direct `call` instruction to its target function address is a direct control-flow transfer. Fortunately, targets of direct control transfers cannot be arbitrarily controlled by attackers (detailed in §1.3.1), so they are less of concern. On the other hand, indirect transfers through indirect branch instructions including indirect calls, indirect jumps and returns are more dangerous, because their targets may be arbitrarily controlled by attackers. To ensure CFI for indirect branches, they are checked before execution so that their targets are always legal.

CFI is a general approach that can secure both operating system kernels and user-level applications. However, in this dissertation, we limit our discussion to only user-level program protection without loss of generality. In addition, we always use x64 (short for x86-64) Linux by default when discussing CFI implementations.

1.3.1 Threat Model

When used to protect user-level applications, CFI assumes that the underlying hardware and software stack, including the firmware, the virtual machine monitor (if there is one), and the operating system kernel, are in the *Trusted Computing Base (TCB)*. CFI requires a CFI-aware program loader (also called the CFI runtime or just runtime) to load modules and generate the CFG, and it is assumed to be trusted. Programs protected with CFI should be built by a CFI-aware compilation toolchain including the compiler, assembler, and linker, and we also consider them trusted.

At runtime, the above trusted modules should enforce a critical invariant: any virtual memory page that contains code or read-only data is never writable. The reason is straightforward: without this invariant, the program could be induced to modify itself and execute arbitrary instructions. (CFI support for self-modifying code needs special consideration, which will be fully discussed in §4.) Fortunately, modern hardware and OSes provide primitives that can enforce this invariant. With this invariant, we consider that direct branches always obey the CFG policy since their targets are statically computed by the trusted compilation toolchain and hard-coded in non-modifiable code.[1] Therefore, CFI is all about protecting indirect branches.

[1] A separate verifier can be built to check direct control transfers, as shown in [1]. We ignore the verification for statically compiled code in this thesis.

We model the attacker as a thread running in the address space of the victim application process. Therefore, the attacker has full control over all writable memory pages allocated to the application. However, the attacker cannot directly modify other application threads' critical registers (e.g., the program counter). Instead, they may somehow change memory values that will be loaded to registers by the application threads themselves, and such changes can even be conducted between any consecutive instructions executed by the application threads.

1.3.2 The Classic Implementation of CFI

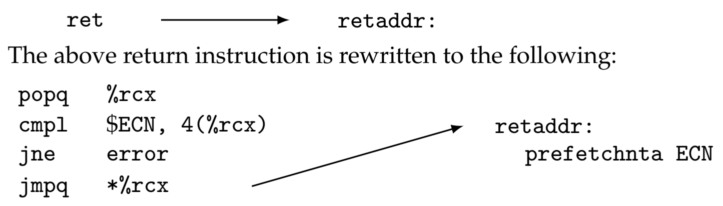
Since CFI's inception, many implementations have been proposed. We next describe the classic (the first) CFI enforcement technique (which we refer to as ClassicCFI) given by Abadi *et al.* [1]. ClassicCFI generates the CFG using a flow-insensitive analysis that conservatively allows any indirect call to target any function whose address is taken. For an indirect branch, ClassicCFI inserts runtime checks, which are described below, before the indirect branch to ensure that the control transfer is always consistent with the CFG.

First, the set of indirect branch targets is partitioned into *equivalence classes* after CFG generation. Two target addresses are equivalent if there exists an indirect branch that can jump to both targets according to the CFG. An indirect branch is allowed to jump to any destination in the same equivalence class. If two indirect branches target two sets of destinations and those two sets are not disjoint, the two sets are merged into one equivalence class.

After partitioning, all addresses in each equivalence class are assigned an *equivalence class number* (ECN), which is a natural number that uniquely identifies the equivalence class. The *branch ECN* of an indirect branch refers to the ECN of the equivalence class whose addresses the branch can jump to. The *target ECN* of an indirect branch is a dynamic notion and refers to the ECN of the equivalence class in which the attempted destination is when the indirect branch runs in a specific state.

For an indirect branch to respect the CFI policy, its branch ECN must be the same as its target ECN. This is enforced by the ClassicCFI instrumentation. Figure 1.3 shows the instrumentation [2] of a return instruction that targets the address `retaddr`:

[2]We have ported 32-bit x86 code from ClassicCFI to x64.



```
    ret        ────────►   retaddr:
```

The above return instruction is rewritten to the following:

```
popq   %rcx
cmpl   $ECN, 4(%rcx)                retaddr:
jne    error                          prefetchnta ECN
jmpq   *%rcx
```

Figure 1.3: An example of the ClassicCFI instrumentation for x64 Linux.

(a) At the target, `retaddr`, ClassicCFI inserts a side-effect-free instruction (`prefetchnta` for memory prefetching) with the target address's ECN embedded in.

(b) The return instruction is rewritten to (1) pop the return address to a temporary register (`%rcx` in this case); (2) retrieve the target ECN and compare it with the branch ECN using `cmpl` and `jne` (the address `%rcx+4` points to the middle of `prefetchnta`, if the correct return address is on the stack); (3) if they are the same, the control is transferred.

In this technique, ECNs are embedded in the non-writable code section to prevent corruption. However, ClassicCFI does not support secure and thread-safe ECN changes, which may be necessary when a new CFG is generated to replace the old one during dynamic code linking. How to design a CFI scheme to support dynamic code linking is one of the challenges this dissertation addresses. We will discuss the details later.

1.3.3 Granularity of CFI

CFI forbids all control-flow transfers not included in the CFG, as a result, the CFG should be *sound*, which means if any edge is required by normal execution, it must appear in the generated CFG. Therefore, a program may have different sound CFGs[3], each of which probably contains redundant edges that are not needed by normal execution.

For each sound CFG of a program, indirect branches and their targets can be partitioned into *equivalence classes* as in ClassicCFI. Indirect branches can target any indirect branch targets in the same equivalence class, but none in other equivalence classes. Different CFI techniques support

[3]In general, it is impossible to generate a minimal sound CFG, since CFG generation is essentially an alias analysis problem, which has been shown to be undecidable [6].

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

Coarse-grained CFI supports a *program-independent* 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 [7] return instructions are allowed to target all return addresses, which are addresses following call instructions. Example coarse-grained CFI techniques include PittSFIeld [8], NaCl [9, 10], CCFIR [11], binCFI [7], and MIP [12]. The major benefit of coarse-grained CFI is that coarse-grained CFGs are easier to build, even without access to source code (e.g., [13]). However, recent attacks presented in [14–16] show that arbitrary computation can be easily constructed without breaking coarse-grained CFI.

Fine-grained CFI

Fine-grained CFI supports a *program-dependent* number of equivalence classes. Each indirect branch can have its own target set. Example fine-grained CFI approaches include several systems [17–21]. Fine-grained CFI provides better security than coarse-grained CFI, because in general, the finer the CFG is, the fewer edges attackers can use, and therefore the harder it is for attackers to mount attacks. Unfortunately, existing fine-grained CFI has several issues that affect its practicality, which will be fully described next.

Based on the above discussion, we can see that coarse-grained CFI is a specialized form of CFI, while fine-grained CFI is general, therefore in the remainder of this dissertation we use the terms "CFI" and "fine-grained CFI" interchangeably.

1.4 Practical Issues of Previous CFI

Despite (fine-grained) CFI's efficacy, it has not seen wide industrial adoption. We believe that not well supporting the following four critical features contributes to CFI's poor deployment:

- **Fine-grained CFGs**. Existing fine-grained CFG generation techniques such as SafeDispatch [22] and ForwardCFI [23] only consider precise edges for C++ virtual method calls, but not others. HyperSafe [18] generates fine-grained CFGs for C programs, but not for C++. There has been no fine-grained CFG generation approach that works for all indirect branches in both C and C++ programs.

  Moreover, all existing CFG generation methods statically generate CFGs for all inputs, but programs may only need a small portion of the edges in a specific run. Therefore, there might be redundant edges in any statically generated CFG from a concrete input's perspective. If we can generate per-input CFGs, the CFG precision could further be improved.

- **Modularity**. Modularity refers to the capability that two program modules can be separately compiled (and instrumented), and linked either statically or dynamically. For example, mainstream operating systems support dynamically linked libraries (DLLs) that can be loaded and linked to a running executable on demand at runtime, but compiled separately from the main executable. This facilitates software development and update, which can then be performed separately. As another important example, Just-In-Time (JIT) compilers for high-level languages such as JavaScript play a key role in today's computing, because they provide a good balance between development and efficiency. For performance, JIT engines emit native code on-the-fly and directly execute the code for performance. However, previous fine-grained CFI approaches do not support modularity, since they do not support safe dynamic code loading. As a result, no JIT compiler can be secured by previous fine-grained CFI.

- **Efficiency**. CFI-protected programs require extra execution time and space compared to their native counterparts. For example, at runtime, the CFI checks are executed no matter whether there are attacks, thus the protected programs are in general slower than the native versions. Furthermore, the checks consume disk and memory space. For instance, Classic-CFI reports 16% performance overhead and enlarges the binary size by 8%. Szekeres *et al.* [24] propose that a CFI approach is efficient, so that it is possible to see practical use, only if its performance overhead is less than 10% when measured on compute-intensive programs such as SPEC, and the less the better. We agree upon that estimate. Space overhead is usually less significant than the performance overhead, as CFI only changes the program code, whose memory consumption at runtime is often much smaller than the program data, but it should also be as small as possible.

- **Interoperability**. In practice, software consists of multiple modules produced by different vendors, and it is hard to guarantee that each vendor would adopt CFI with the same pace. In addition, legacy modules may not even have source code available. Therefore, it is crucial to support interoperability, which means that instrumented modules and uninstrumented modules can be linked together and run without breaking the program. With interoperability, vendors and users can roll out their own plans for CFI instrumentation and adoption.

None of the previous CFI approaches meets all of the above four requirements. For example, ClassicCFI only supports fine-grained CFG; Zeng *et al.* ([25], [26]) improve ClassicCFI's performance to make it efficient, but still do not support modularity. The reason is that ClassicCFI does not allow the running module's CFG to be combined with other modules' CFGs at runtime, because ClassicCFI embeds all IDs in the code region and cannot safely update them to reflect new equivalence classes in the combined CFG.

1.5 Challenges to CFI Practicality

Apparently, a CFI approach supporting all the four features is more practical than all existing CFI methods, as it provides better security due to fine-grained CFGs, conforms to programming convention because of modularity and interoperability, and incurs less performance overhead thanks to high efficiency. However, designing such a CFI approach is not without general challenges:

- **Fast combination of fine-grained CFGs.** When two modules each of which carries a fine-grained CFG are linked at runtime, their CFGs should be combined to form a CFG for the linked modules[4]. Since the CFGs are combined at runtime, the combination should be as

[4]CFG generation for a single module is a special case when the module is linked with an empty module.

fast as possible, otherwise the efficiency feature is lost. Therefore, we cannot afford expensive online analysis to extract the combined CFG when modules are linked; instead, fast and scalable approaches are preferred. Further, without expensive (usually comprehensive) static analysis, we might lose precision of CFGs to some extent. However, such precision loss should be mild and therefore still leads to fine-grained CFGs.

- **Atomic query and update of the enforced CFG.** After the running module's CFG is combined with the loaded module's CFG at runtime, the enforced CFG on the running module has probably changed but should still be sound. However, during the CFG update, there might be other threads running, whose CFI checks consult the CFG for capturing CFI violations. Unfortunately, these threads may access certain unsound intermediate CFGs if the CFG query or update is not atomic. For example, suppose before module linking, two indirect branches b_1 and b_2 can target memory addresses A_1 and A_2, respectively, and after module linking, b_1 should target A_1, A_3 and b_2 should target A_2, A_4. One possible intermediate CFG would allow b_1 to target A_1, A_3 and b_2 to target only A_2 but not A_4. This unsound CFG breaks the program semantics by disallowing b_2 to reach A_4. Consequently, the CFG query and update operations should both be atomic to ensure that each thread only sees sound CFGs. Although such atomicity can be naïvely implemented using locks in the inserted CFI checks, they are expensive instructions, which will jeopardize the efficiency.

- **Fast execution of CFI checks.** CFI checks are inserted right before frequently executed instructions such as function returns. As a result, the checks should be as efficient as possible so that the execution slowdown is tolerable.

- **Interoperability with acceptable CFI protection weakening.** When uninstrumented modules are linked to instrumented modules, the CFI protection for instrumented modules could be degraded but should never disappear. Basic CFI protection (coarse-grained) should still be preserved for instrumented modules.

1.6 Thesis Statement

Given these challenges, an interesting question is: does a practical CFI solution that supports fine-grained CFGs, modularity, efficiency and interoperability exist?

Thesis Statement: *Control-Flow Integrity can be fine-grained, modular, efficient and interoperable, and therefore practical.*

1.7 Contributions

The thesis statement is fully supported by the following contributions described in this dissertation.

1. Two fine-grained, modular, efficient and interoperable approaches to CFI. One approach, dubbed Modular Control-Flow Integrity (MCFI), solves the above basic problem. The other approach, entitled Per-Input Control-Flow Integrity (PICFI or πCFI), builds upon MCFI and provides even finer-grained CFGs customized for concrete program inputs.

2. A framework that can instrument and load programs using the above CFI approaches and securely execute CFI-protected programs. We changed the Clang/LLVM compilation toolchain for program instrumentation, and rolled out a user-level CFI-aware runtime, running in x64 Linux. The toolchain is open source and hosted at `https://github.com/mcfi`.

3. A general approach entitled RockJIT to extend MCFI and πCFI to supporting JIT compilers. We ported the Google V8 JavaScript engine to demonstrate the protection.

4. A study on the practicality of the above proposed CFI. Using the above tools, we instrumented and measured CFG precision, modularity, efficiency and interoperability on a variety of programs.

1.8 This Dissertation versus Previous Publications

The content of this dissertation, especially Chapter 2, 3 and 4, is based on our previous publications [27–29]. However, we make improvements by presenting technical details uncovered in the published papers, adding more experimental results, etc. The detailed differences can be found in §2.8, §3.6 and §4.7.

1.9 Outline

The remainder of this dissertation is organized as follows: Chapter 2 describes Modular Control-Flow Integrity. Chapter 3 presents Per-Input Control-Flow Integrity. Chapter 4 discusses RockJIT. Chapter 5 presents our case-by-case security analysis. Chapter 6 elaborates some related work. Chapter 7 draws conclusions.

Chapter 2

Modular Control-Flow Integrity

2.1 Overview

Modular Control-Flow Integrity (MCFI) is the first CFI approach that supports fine-grained CFGs, modularity, efficiency and interoperability. To generate a fine-grained CFG, MCFI propagates source-level information such as type information to the binary level as metadata, and gathers such metadata at program load time to build a precise CFG, which is consulted (or read) by the program to detect CFI violations. When a code module is loaded during execution, the loading module's metadata is combined with the loaded module's metadata to compute a CFG for both modules. The old CFG will then be replaced with the new CFG in order not to break program execution. Since the CFG update operation might be executed by a concurrent thread, while other threads may be reading it, data races may occur. To resolve the race condition, we designed a lightweight Software Transactional Memory (STM) scheme to synchronize the CFG query and modification operations, which results in small performance overhead. When uninstrumented modules are linked, since no metadata is available, we conservatively mark each code address in the uninstrumented modules as a legal target for any indirect branch, and we revise the transaction scheme to appropriately handle this special case. In addition, the fine-grained CFG generated for all instrumented modules is coarsened to support interoperability.

2.2 Fine-grained CFG Generation

MCFI is designed for enforcing CFI on binaries compiled from C or C++. Since the C language is a subset of C++, we focus our discussion on C++. In practice, C/C++ programs may be mixed with assembly code, and we also discuss how to handle CFG generation for assembly. C++ CFG generation depends on the Application Binary Interface (ABI), and we focus on the Itanium ABI [30] used in x64 Linux.

2.2.1 Source-Level Semantics-based CFG Generation

MCFI needs to generate CFGs (or specifically indirect edges as discussed) for program binaries compiled from C++. For each indirect branch instruction (e.g., an indirect call) and each indirect branch target (e.g., a return address), MCFI propagates its source-level information (e.g., virtual call function type) to metadata packaged with the binary. At module load time, such metadata is extracted from each module and combined to generate the CFG. Next, we analyze all possible C++ constructs (e.g., virtual method call) that might be lowered to an indirect branch instruction and for each case how to generate edges for the indirect branch.

Virtual Method Calls

C++ supports multiple inheritance and virtual methods. A virtual method call through an object is usually compiled to an indirect call (or an indirect jump with tail call optimization). A virtual call on an object is resolved during runtime through dynamic dispatch. Which method it invokes depends on the actual class of the object. Similar to SafeDispatch [22], MCFI performs Class Hierarchy Analysis (CHA) [31] on C++ code. This analysis tracks the class hierarchy of a C++ program and determines, for each class C and each virtual method of C, the set of methods that can be invoked when calling the virtual method through an object of class C; these methods might be defined in C's subclasses. MCFI simply allows a virtual method call to target all methods determined by CHA. Also, note that to support C++ multiple inheritance, the compiler may generate thunks [32], which are simple trampolines that first adjust "`this`" pointer [33] and then jumps to the corresponding virtual methods. The thunks may be used to fill virtual tables instead

```
 1 class A {
 2 public:
 3   virtual void foo() const {}
 4   virtual void foo() {}
 5   virtual void bar() const {}
 6   virtual void bar() {}
 7 };
 8 class B : public A {
 9 public:
10   virtual void foo() const {}
11   virtual void foo() {}
12   virtual void bar() {}
13 };
14 ...
15 void fx(A *a) {
16   a->foo();
17 }
18 void fy(B *b) {
19   b->foo();
20 }
```

Figure 2.1: A toy C++ example of virtual method call targets.

of their corresponding virtual methods and therefore can be called by virtual method invocation as well. We associate each thunk with the same meta-information as its corresponding virtual method and add it to the class hierarchy as well for CFG generation.

Next we use a toy C++ example in Figure 2.1 to demonstrate the basic idea. We define a class `A` and its subclass `B` as well as their virtual methods. In function `fx`, virtual method `foo` is invoked with respect to a class `A` object pointer; in function `fy`, `foo` is invoked with a class `B` object pointer. According to the class hierarchy, which can be constructed straightforwardly, `a->foo()` at line 16 possibly targets `A::foo` and `B::foo`, while `b->foo()` at line 19 can target `B::foo`. Note that `a->foo()` should not reach `A::foo const` at line 3 or `B::foo const` at line 10, because their type qualifiers [34] do not match: `a->foo()` calls a non-constant virtual method, but `A::foo const` and `B::foo const` are constant methods.

It should be pointed out that CHA is a whole-program analysis. To support modularity, MCFI emits a class hierarchy for each module and combines modules' class hierarchies at link time.

```
1       typedef int (*Fp)();
2       Fp fp = &getpagesize;
3       std::cout << (*fp)();
4       ...
5       typedef void (A::*memFp)() const;
6       ...
7       void fz(const A *a) {
8         memFp memfp = &A::foo;
9         (a->*memfp)();
10      }
```

Figure 2.2: An example about C++ function pointers.

Function Pointer Dereference

C++ supports two kinds of function pointers: (1) those that point to global functions or static member methods; (2) those that point to non-static member methods. Function pointers in these two kinds have different static types. Their target sets are disjoint and they are handled differently by compilers. Figure 2.2 shows a code example about the two kinds of function pointers.

Function pointer `fp` is of the first kind. It is assigned to the address of a global function `getpagesize` at line 2. At line 3, the function pointer is invoked via an indirect call (or indirect jump if it is a tail call). To identify its targets, MCFI adopts a type-matching method: an indirect branch via a function pointer of type $\tau*$ can target any global function or static member method whose static type is equivalent to τ and whose address is taken in the code. For simplicity, we use Clang to compile C/C++ programs to LLVM intermediate representation (IR) and use the names of LLVM IR types to represent function types. Two types are equivalent if and only if their names are literally the same. Taking a function's address means that the function's address is assigned to a function pointer somewhere in the code (e.g., line 2).

Function pointer `memfp` at line 9 is of the second kind, which is also called a method pointer. The code reuses the class definition in Figure 2.1. According to the C++ semantics, we allow an indirect branch through such a method pointer of type $\tau*$ to target any virtual or non-virtual member method defined in the same class whose type is equivalent to τ, whose address is taken and whose type qualifier such as `const` matches the method pointer's. (LLVM IR does not support function type qualifiers, so we changed Clang and LLVM to propagate C++ member method type qualifiers to the LLVM IR as metadata.) Moreover, for each matched virtual member method, we search the class hierarchy to find in derived classes all virtual methods whose types and qualifiers match and add those functions to the target set, because, for example, if a `B` object pointer is passed at line 7, `B::foo const` will be called. Consequently, the method pointer dereference `(a->*memfp)()` at line 9 can possibly reach `A::foo const`, `A::bar const` or `B::foo const` in Figure 2.1 at line 3, 5 or 10, respectively.

It should be noted that function addresses can also be explicitly taken at runtime by libc function `dlsym`. Therefore, we changed `dlsym`'s implementation so that before `dlsym` returns a valid function address, an MCFI runtime trampoline (details in §2.3.2) is called to mark the function's address as taken and update the CFG so that function pointers with the equivalent type can legitimately call the function. Later in §3, we show that by carefully handling address-taken events, we can further refine the CFGs.

Returns

To compute control-flow edges out of return instructions, we construct a call graph, which tells how functions get called by direct or indirect calls, which have been described above. Using the call graph, control-flow edges out of return instructions can be computed: if there exists an edge from a call node to a function, return instructions in the function can return to the return address following the call node.

In addition, modern compilers usually perform tail-call elimination at machine code level to save stack space. Basically, if a return instruction immediately follows a call instruction during code emission, the return is eliminated and the call is replaced with a jump. We handle this case in the following way: if in function `f`, there is a call node calling `g`, and `g` calls `h` through a series of tail jumps, then an edge from the call node in `f` to `h` is added to the call graph. Unfortunately, tail-call elimination may introduce CFG precision loss, and we discuss the problem in detail in §2.2.3.

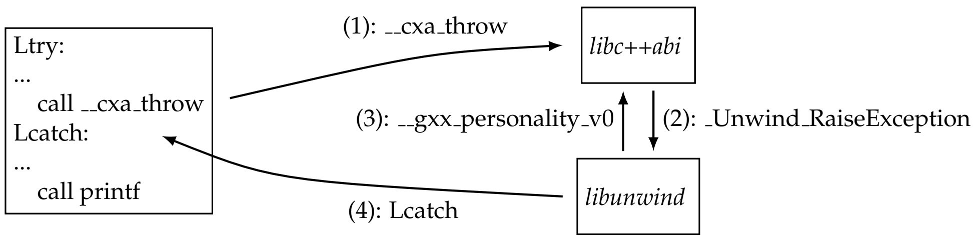


Figure 2.3: Control transfers during C++ table-based exception handling.

Exception Handling

We first discuss how C++ exceptions are handled by compilers and libraries that implement the Itanium C++ ABI. In this ABI, C++ exception handling is joint work of the compiler, a C++-specific exception handling library such as `libc++abi` and a C++-agnostic stack-unwinding library such as `libunwind`.

When a compiler compiles a C++ program, it emits sufficient information for stack unwinding, since every stack frame needs to be searched to find a matching `catch` clause for a thrown exception object. Such data is emitted as metadata (e.g., the `eh_frame` and `gcc_except_table` sections in an ELF file) during compilation. Figure 2.3 depicts the runtime control flow when an exception object is thrown. It assumes `libc++abi` and `libunwind` are used; the control flow would be the same when other libraries are used as long as they obey the Itanium C++ ABI.

The left box in Figure 2.3 shows some assembly code, where the `Ltry` label starts a C++ `try` statement and `Lcatch` implements a `catch` statement. A C++ `throw` statement is translated to a direct call to `libc++abi`'s `__cxa_throw`, which takes three arguments: the heap-allocated exception object, its type information, and a pointer to the object's destructor. It performs initialization and invokes `_Unwind_RaiseException` in `libunwind`, which extracts the code address where the exception is thrown and walks through each stack frame by consulting the `eh_frame` section. In each stack frame, `_Unwind_RaiseException` uses an indirect call to invoke a C++-specific routine called `__gxx_personality_v0` defined in `libc++abi`, which searches for catch clauses in that frame by consulting `gcc_except_table`. Two cases can happen. If a type-matching catch clause is found in the current frame, control is transferred to the catch clause via an indirect branch, which we call `CatchBranch`. If a type-matching catch is not found, the stack unwinding should be resumed.

However, if there is a clean-up routine that is used to deallocate objects allocated in `try` statements, the clean-up routine needs to run before the unwinding continues. It turns out that the same indirect branch (`CatchBranch`) is used to transfer the control to the clean-up routine, but with a different target address.

Consequently, all control-flow edges in Figure 2.3, except for the edges out of `CatchBranch`, can be handled using the strategies we have discussed (CHA and the type-matching method). For the `CatchBranch`, our implementation connects it to all catch clauses and cleanup routines. To support separate compilation, MCFI's modified LLVM compiler emits a table recording addresses of all catch clauses and cleanup routines in each module, and these tables are combined during linking.

If an exception object is caught, but not rethrown, `libc++abi` invokes the object's destructor, which is registered when calling `__cxa_throw`. The invocation is through an indirect call. Possible targets of this call in a module can be statically computed by tracking `__cxa_throw` invocations. As a result, MCFI's compiler emits these target addresses for each module and the runtime combines them at link time.

Global constructors and destructors. The constructors of global and local static objects are invoked before the main function of a C++ program, and their destructors are called after the main function returns. LLVM handles such cases by generating stub code for each such object. The stub code directly invokes the constructor and registers the destructor using either `__cxa_atexit` or `atexit` defined in `libc`. The addresses of the stub code are arranged in the binary and iterated by an indirect call (called *CtorCall*) in `libc` before `main`. After `main`, another `libc` indirect call (called *DtorCall*) iterates the registered destructors to destroy objects. Both CtorCall and DtorCall's targets are statically computable by analyzing the compiler-generated stub code.

Signal Handlers

In Linux, signal handlers are usually not invoked by any application code[1], so they do not return to the application code. Instead, signal handlers return to a code stub set up by the OS kernel,

[1]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.

```
memcpy:
  ...       # instructions omitted
__mcfi_return_memcpy:
  mcfi-ret # MCFI-instrumented return

  .section MCFIMetadata,"",progbits
  .ascii "memcpy : void* (*)(void*, void*, size_t)"
```

Figure 2.4: Metadata annotation for the assembly code of `memcpy`.

which invokes the `sigreturn` system call. MCFI provides new function attributes for developers to annotate signal handlers in the source code so that the compiler will inline the code stub into each signal handler during code generation. Each signal handler is associated with a special type signature to ensure it never becomes any indirect call target. This design helps mitigate Sigreturn-Oriented Programming attacks [35], which will be discussed in §5.1.6.

Indirect Control-Flow in Assembly Code

Indirect branches and indirect branch targets in assembly code should be appropriately handled to enable CFG edge generation with MCFI-instrumented C/C++ code. MCFI requires the developers to manually annotate the assembly instructions so that the assembly code seems like being compiled by the MCFI compiler. For example, some libc functions such as `memcpy` is implemented using manually written assembly for performance, and suppose Figure 2.4 shows `memcpy`'s starting label and the instrumented return instruction `mcfi-ret` (the instrumentation is later explained in §2.3). To help generate the CFG, type information needs to be added for the assembly function of `memcpy`. Moreover, its instrumented return instruction should be annotated so that the MCFI runtime knows which indirect branch in the binary code performs the function return operation for `memcpy`. To achieve these, we insert an MCFI-specific label "`__mcfi_return_memcpy`" for identifying `memcpy`'s instrumented return, and add a string (enclosed in double quotes) in a newly created section "`MCFIMetadata`" to record the type information. It should be noted that for simplicity, the annotation format in Figure 2.4 is slightly different from the format used in our actual implementation.

```
1       typedef void (*F)(char*); /* define F to be void (*)(char*) */
2       void foo(long x); /* a global function */
3       ...
4       F fp = (F)&foo; /* cast the type of foo to F */
5       (*fp)(0); /* False CFI violation */
```

Figure 2.5: An example showing that our CFG generation process may break C code with type casts from function pointers.

Other Control-Flow Features

According to the specification, `longjmp` always returns (through an indirect jump instruction) to the address set up by a `setjmp` call. MCFI simply connects the `longjmp`'s indirect jump to the return addresses of all `setjmp` calls. Other functions such as `setcontext` and `getcontext` can be handled in a similar way.

Switch and indirect `goto` statements are typically compiled to direct jumps or jump-table based indirect jumps; their targets are statically computed and embedded in read-only code or jump tables, so they do not need instrumentation.

Lambda functions are available in C++11, whose related control-flow edges (returns) are also supported by our CFG generation. Compilers automatically convert lambda functions to functors, which are classes with `operator()` methods that are directly called. The return edges of the `operator()` methods can be handled in the same way as those of other functions.

2.2.2 CFG Soundness

Due to arbitrary type cast, our aforementioned CFG generation approach may not generate a sound CFG for an arbitrary C/C++ program because of the indirect branch edge analysis for virtual method calls and function pointer dereferences. Consider a toy C example in Figure 2.5. The function pointer dereference at line 5 would be falsely detected as a CFI violation, because the function pointer `fp`'s type is `void (*)(char*)`, but it actually points to function `foo` that has a different type, which means the edge that connects `fp` to `foo` is missing in the generated CFG.

We believe that MCFI can generate a sound CFG for a *memory-safe* C/C++ program if the program satisfies a *compatibility condition* that *it has no bad type cast from or to type* T *(defined below)* *that contains function pointer types*, because each indirect call will only be assigned values of the matching type. Here, T is either a function pointer type, or an aggregate type (e.g., struct) that has a field of type T, or a pointer type pointing to an object of type T. Note that C++ classes with virtual methods are also of type T as they essentially contain a virtual table pointer pointing to virtual method pointers. Moreover, note that MCFI allows each virtual method call to reach any virtual method implementation according to the class hierarchy, so the condition can be relaxed to allow all type casts between two classes (or class pointers) as long as there exists an inheritance path in the class hierarchy between those two classes.

In practice, not every C/C++ program satisfies the above condition, so they first need to be retrofitted to meet the condition. We built a Clang-based static checker that can capture all bad casts to facilitate the retrofitting process. The checker is executed after the Abstract Syntax Tree (AST) is generated in Clang, which explicitly represents all type casts.

Effort to Retrofit Existing C/C++ Code

We investigated how much effort it takes to make SPECCPU2006 C/C++ programs comply with the compatibility condition. First we present the results for the twelve C programs in Table 2.1. For a benchmark, column "SLOC" lists its lines of source code and column "VBE" (Violation Before false-positive Elimination) lists the number of condition violations. While some benchmarks such as `445.gobmk` have no violations, two benchmarks, `400.perlbench` and `403.gcc`, have thousands of violations. We found many cases do not lead to actual violations of the CFG built by our system; that is, they are false positives.

Some of the false positives have common patterns and can be easily ruled out by the analyzer. We next briefly discuss those cases: (1) *Upcast (UC).* C developers sometimes use type casts between structs to emulate features such as parametric polymorphism and inheritance. An abstract struct type is defined and it contains common fields for its subtypes. Then, a few concrete struct types are physical subtypes of the abstract struct type (in the sense that they share the same prefix of fields). A function can be made polymorphic by accepting values of the abstract struct type. Callers of the function have to perform type casts. Those type casts are upcasts, which are false positives in our system because the extra fields in a concrete struct cannot be accessed after the

SPECCPU2006	SLOC	VBE	UC	DC	MF	SU	NF	VAE
400.perlbench	126,345	2878	510	957	234	633	318	226
401.bzip2	5,731	27	0	0	6	4	0	17
403.gcc	235,884	1366	0	0	15	737	27	587
429.mcf	1,574	0	0	0	0	0	0	0
445.gobmk	157,649	0	0	0	0	0	0	0
456.hmmer	20,658	20	0	0	20	0	0	0
458.sjeng	10,544	0	0	0	0	0	0	0
462.libquantum	2,606	1	0	0	0	0	0	1
464.h264ref	36,098	8	0	0	8	0	0	0
433.milc	9,575	8	0	0	3	0	0	5
470.lbm	904	0	0	0	0	0	0	0
482.sphinx3	13,128	12	0	0	11	1	0	0

Table 2.1: Condition violations in SPECCPU2006 C benchmarks.

cast. (2) *Safe downcast (DC).* Downcasts from an abstract struct type to a concrete struct type are in general not safe. However, a common pattern is to have a type tag field in the abstract struct; the runtime type tag encodes the type of a concrete struct when it is cast to the abstract struct. Clearly, if all casts involving the abstract struct type respect a fixed association between tag values and concrete struct types, those casts can be considered false positives. Such association can be specified manually (or inferred from source code) and fed to the analyzer. (3) *Malloc and free (MF).* `malloc` always returns `void*`. If it is invoked to allocate a struct that contains function pointers, the compatibility condition is violated as it involves a type cast from `void*` to a struct with function pointers inside. We consider such violations false positives because if the function pointers inside the struct are used without proper initialization, the C program is not memory safe. Similarly, type casts in invocations of `free` are also considered false positives. (4) *Safe update (SU).* We consider updating function pointers with literals as false positives. For instance, function pointers may be initialized to be `NULL`, which involves a cast from integers to function pointers. This is a false positive as dereferencing a null value would terminate the program. (5) *Non-function-pointer access (NF).* There are some type casts that involve function pointers but after casts the function pointers are not used. Take the following example from `400.perlbench`.

```
  if (((XPVLV*)(sv->sv_any))->xlv_targlen) { ... }
```

Struct `XPVLV` has a function-pointer field, but after the cast only non-function-pointer fields are used. It is a false positive.

	400.perlbench	401.bzip2	403.gcc	462.libquantum	433.milc
K1	4	0	580	1	0
K1-fixed	4	0	72	1	0
K2	222	17	7	0	5

Table 2.2: Numbers of cases for the two kinds of violations.

In Table 2.1, columns "UC", "DC", "MF", "SU" and "NF" list the numbers of false positives removed by our aforementioned elimination methods. Column "VAE" presents the number of cases after elimination. As can be seen from the table, the elimination methods are effective at eliminating a large number of false positives. After the process, seven C benchmarks report no violations and need no code fixes. For the other five C benchmarks, the remaining cases can be put into the following kinds:

K1 A function pointer is initialized with the address of a function whose type is incompatible with the function pointer's type.

K2 A function pointer is cast to another type and cast back to its original type at a later point.

Table 2.2 reports the number of K1 and K2 cases in the remaining five benchmarks. Row "K1-fixed" lists the number of cases in K1 that require changes to the source code to generate a working CFG using the type-matching method. None of the cases in K2 requires us to change the source code.

Most K1 cases require us to change the source code manually because the unmatched types of function pointers and functions may cause missing edges in the generated CFG. Consider a case in the `403.gcc` benchmark that is related to a generic splay tree implementation. Each node in the splay tree has a key typed `unsigned long`. There is a key-comparison function pointer typed `int (*)(unsigned long, unsigned long)`. In two places, the function pointer is set to be the address of `strcmp`, whose type is `int (*)(const char*, const char*)`. Since the function pointer's type is incompatible with `strcmp`'s, the CFG generation does not connect the function pointer to `strcmp`. To fix the problem, we added a `strcmp` wrapper function that has the equivalent type as the type of the comparison function and makes a direct call to `strcmp`. The key-comparison function pointer is then set to be the address of the wrapper function. For another

Program	SLOC	K1	K1-fixed	K2	VAE
444.namd	3,886	0	0	0	0
447.dealII	94,384	0	0	15	15
450.soplex	28,277	0	0	0	0
453.povray	78,705	35	35	25	60
471.omnetpp	19,991	0	0	48	48
473.astar	42,80	0	0	0	0
483.xalancbmk	267,399	0	0	350	350

Table 2.3: Condition violations in SPECCPU2006 C++ benchmarks.

example in `403.gcc`, different instruction emission functions have different numbers of parameters, and `403.gcc` initializes a variadic function pointer array with those instruction emission functions' addresses, which contribute most K1 cases. To fix those cases, we cast the variadic function pointer to a non-variadic function pointer at its call sites. All cases in the "K1-fixed" row can be fixed by wrappers or by directly changing the types of function pointers or functions.

Four benchmarks report K2 cases. Consider an example in the `400.perlbench` program. A function pointer is initially stored in a `void*` pointer and later the `void*` pointer is cast back to the original function pointer's type and dereferenced. In `400.perlbench` and `403.gcc`, there are also cases of downcast without performing dynamic checking on type tags. In these cases, developers decided those downcasts are safe (perhaps through code inspection) to avoid dynamic checks. None of the K2 cases required code changes to generate a working CFG. This was confirmed by running instrumented benchmarks successfully with all the provided data sets.

Similarly, we ran the analyzer on the seven C++ benchmark programs and found three of them satisfy the compatibility condition while six of them do not need any code fix. Details are shown in Table 2.3

Our experience on SPECCPU2006 shows that the task of making source code work with the type-matching approach is not onerous and can be achieved without many changes to the code. Furthermore, our empirical investigation suggests that only K1 cases are the ones that need fixes.

2.2.3 CFG Precision Loss

As discussed, the tail-call elimination optimization during machine code emission may be performed to replace a call followed by a return with a jump, which can save stack space and may

```
 1 // (a) no tail-call elimination   // (b) tail-call elimination
 2 f:                                f:
 3   call foo                          call foo
 4 L:                                L:
 5   ...                               ...
 6 foo:                              foo:
 7   ...                               ...
 8   call bar                          jmp bar
 9   ret
10 bar:                              bar:
11   ...                               ...
12   ret                               ret
```

Figure 2.6: Example of CFG precision loss due to MCFI tail-call elimination.

speed up program execution. However, this optimization introduces CFG precision loss by enlarging target sets of return instructions, as demonstrated by the example code in Figure 2.6. Figure 2.6 (a) shows the code without tail-call elimination, and the above described CFG generation method will allow function `foo`'s return instruction at line 9 to target label `L`, but not `bar`'s return. Figure 2.6 (b) lists the code after tail-call elimination, which connects function `bar`'s return to `L` as well as all return addresses following call sites to `foo`, thus resulting in a larger return address set and CFG precision loss. Fortunately, such CFG precision loss can be recovered by disabling the instruction replacement, and the recovered CFG can be enforced by the MCFI instrumentation (detailed in §2.3).

After the CFG is generated, MCFI performs equivalence class partitioning and assigns a unique ECN to each indirect branch and indirect branch target, same as ClassicCFI (details in §1.3.2). However, this process might result in CFG precision loss as well, because the finally enforced CFG is coarser-grained than the generated CFG, in the following cases.

Return addresses may be merged into an equivalence class due to indirect calls, and Figure 2.7 shows an example. According to the CFG generation strategy, function `foo`'s return (omitted in the figure) should target `Lfoo` or `L`, and `bar`'s return (also omitted) should target `Lbar` and `L`. However, the equivalence class partitioning process will merge `Lfoo`, `Lbar` and `L` into the same equivalence class because all of the three addresses share the same ECN.

Moreover, C++ virtual method calls may legally reach more virtual methods defined in super

```
  call foo
Lfoo:
  ...
  call bar
Lbar:
  ...
  call *%rax # rax may be equal to foo or bar
L:
  ...
```

Figure 2.7: Example of CFG precision loss because of indirect calls.

classes. According to the class hierarchy of Figure 2.1, the virtual method call `b->foo()` at line 19 should only target `B::foo`. However, since the virtual method call `a->foo()` at line 16 can reach both `A::foo` and `B::foo`, `B::foo` will be assigned with the same ECN as `A::foo`, essentially allowing `b->foo()` to also target `A::foo` in the enforced CFG.

Finally, C++ method pointers can cause CFG precision loss and let us take the C++ code in Figure 2.8 as an example. According to the CFG generation approach mentioned in §2.2.1, `a->foo` at line 11 should only reach `A::foo`, while `a->*memfp` at line 12 can target either `A::foo` or `A::bar`. However, since MCFI assigns the same ECN to both `A::foo` and `A::bar` (as well as `a->foo` and `a->*memfp`), `a->foo` can also target `A::bar` at runtime. The effect is equivalent to the merging of the equivalence classes of `A::foo` and `A::bar`, which should have been different without the method pointer.

Unfortunately, the equivalence class partitioning process is required by the MCFI instrumentation discussed next, so the CFG loss caused by this process cannot be recovered and enforced by the instrumentation.

2.3 Modularity and Efficiency

After the fine-grained CFG is generated, MCFI partitions indirect branch targets into equivalence classes and labels each with an ECN, same as what ClassicCFI does. To remove the global uniqueness requirement in ClassicCFI, ECNs are pulled out of the code section and stored in a runtime

```
class A {
public:
  virtual void foo(void) {}
  void bar(void) {}
};

typedef void (A::memFp*)(void);

memFp memfp = &A::bar;
A *a = new A;
a->foo();
(a->*memfp)();
```

Figure 2.8: Example code of CFG precision loss due to C++ method pointers.

data structure consisting of two separate tables. These tables are conceptually maps from addresses to IDs, each of which contains an ECN and other components (detailed later in §2.3.1). The branch ID table, called the *Bary table*, maps from an indirect-branch location to the location's branch ID, which contains the ECN of the equivalence class of addresses the branch is allowed to jump to. The target ID table, called the *Tary table*, maps from an address to an ID showing the equivalence class to which the address belongs.

With the ID tables, instrumenting an indirect branch is straightforward. Take the example of a return instruction located at address `l`. The instrumentation can first use the Bary table to look up the branch ID for address `l`, use the Tary table to look up the target ID for the actual return address, and check whether the branch ID is the same as the target ID.

This CFG encoding has several benefits. First, IDs in the tables can overlap with the numbers in the code section, eliminating the global ID uniqueness assumption in ClassicCFI. Second, the instrumentation code before indirect branches is parameterized over the ID tables and remains the same once loaded. Therefore, code pages for applications and libraries can be shared among processes, saving memory and application launch time.

With specially encoded IDs, we design table access operations as transactions to enable thread-safe table look-ups and updates. The look-up operation does not use any heavy bus-locking instruction, which makes the transaction execution efficient.

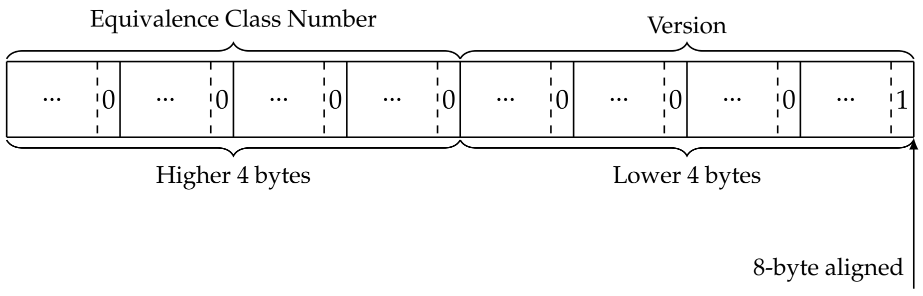


Figure 2.9: MCFI's ID Encoding for x64.

2.3.1 Design of IDs and ID Tables

As discussed, MCFI maintains two tables, both of which map from addresses to IDs. The Bary table holds branch IDs and the Tary table holds target IDs. An ID is eight-byte long, visualized in Figure 2.9. An ID is stored in an eight-byte aligned memory address so that a single x64 memory access instruction can atomically access it.

An MCFI ID contains several components. The first component is composed of the least significant bits in the eight bytes. They are reserved and have the special bit values 0, 0, 0, 0, 0, 0, 0, and 1, from high to low bytes. These reserved bits are to prevent the use of an address that points to the middle of an ID to look up the tables; more on this will be discussed shortly. We define a *valid ID* to be an ID that has the special bit values at the reserved-bit positions.

Besides, an MCFI ID contains a 28-bit ECN in the higher four bytes and a 28-bit version number in the lower four bytes. Our ID-encoding scheme allows 2^{28} different equivalence classes in programs. The version number in an ID is to support table access transactions and is used to detect whether a check transaction should be aborted and retried (details discussed later). The ID encoding allows 2^{28} different version numbers.

We next discuss how MCFI represents Bary and Tary tables during runtime. Since they are queried frequently, MCFI should choose an appropriate data structure to minimize the ID-access time. There is a range of data structures MCFI could use. A naïve choice is a hash map that maps from addresses to IDs. This is space efficient, but the downside is that an ID access involves many instructions for computing the hash value and even more when there is a hash collision.

Instead, MCFI adopts a simple representation of the ID tables. Both Bary and Tary tables are represented using arrays. The Tary table is an array of IDs indexed by code addresses. If a code address is not a possible indirect-branch target, the corresponding array entry contains all zeros; otherwise, it contains the ID of the code address. This design clearly enables efficient look-ups and updates, but one worry is its space efficiency.

In the case that there is an entry in the table for every code address, the size of the table is eight times of the code size since each ID is eight-byte long. To have a smaller Tary table, MCFI uses a space-optimization technique. It inserts extra nop instructions into the program to force indirect-branch targets to be eight-byte aligned. As a result, the table needs entries only for eight-byte aligned code addresses, and the size of the Tary table is the same as the code size. During runtime, since the majority of memory consumed by a program holds data in the heap, the Tary table causes only a small increase on the runtime memory footprint. The alignment also guarantees that any program whose code size is no more than 2GB is supported by MCFI, as there will be no more than 2^{28} indirect branch targets in the code, which is equal to the maximum number of equivalence classes.

Moreover, MCFI has to prevent programs from using indirect-branch targets that are not eight-byte aligned. This is where those reserved bits in an ID help[2]. In particular, if an indirect branch uses an address that is not eight-byte aligned, the eight-byte target ID loaded from the Tary table will not be valid (i.e., it will not have the special bit values 0, 0, ..., and 1 in the least-significant bits). Then, the comparison with the branch ID will fail because the branch ID loaded from the Bary table is always valid, as discussed next.

The Bary table could use the same design as the Tary table, but MCFI uses an optimization to increase its space and time efficiency. Recall that the Bary table conceptually maps indirect-branch locations to branch IDs. One observation is that instruction addresses are known once they are loaded in memory. Therefore, when a module is loaded into the code region, MCFI's loader patches the code to embed constant Bary table indexes that correspond to correct branch IDs in branch-ID read instructions. In this design, the Bary table does not need entries for code addresses that do not hold indirect branches (in contrast, the Tary table has all-zero entries even

[2]Alternatively, we can insert an `and` instruction to align the indirect-branch targets by clearing the least two bits, but it incurs more overhead.

for addresses that are illegal indirect-branch targets). Furthermore, all branch IDs loaded from the Bary table are valid IDs as long as the loader embeds the correct table indexes in branch-ID read instructions.

2.3.2 Memory Layout of MCFI and Protection of ID Tables

The Bary and Tary tables need to be protected at runtime so that application code cannot directly change them. Figure 2.10 shows the memory layout of an application protected with MCFI. The application should have been compiled and instrumented by MCFI's compilation toolchain. The application and all its instrumented libraries are loaded into a sandbox created by the MCFI runtime. The sandbox can be realized using Software-based Fault Isolation (SFI [36]) or hardware support (e.g., segmentation). In our case, we use the scheme described in [37] to create the SFI sandbox. In detail, the sandbox for running applications is within [0, 4GB) [3], and the MCFI 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. As a result, code in the sandbox cannot arbitrarily execute or write memory pages outside the sandbox, but has to invoke trampolines provided by the MCFI 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, according to the threat model of CFI (§1.3.1). In addition, the runtime guarantees that read-only data, such as jump tables, are not writable. Consequently, those system calls that might subvert the invariant are replaced with runtime trampoline invocation. For instance, the `mmap`, `munmap` and `mprotect` system calls in the libc are all rewritten to invoke the relevant runtime trampolines that are checked. The MCFI runtime and the encoded CFG, i.e., Bary and Tary, are stored outside of the sandbox. The ID tables are read-only from the application's perspective, but writable by the runtime.

MCFI uses the `%gs` segment register to index both the Bary and Tary tables. Inside the sandbox, MCFI always loads the code in [4MB, 4GB) to comply with the x64 Linux ABI, and the region

[3]The maximum sandbox size can be extended to 64TB on x64 if the sandboxing technique in PittSFIeld [8] is used or the MCFI runtime is implemented as a kernel module.

[0, 4MB) is always unmapped. Therefore, we allocate [%gs+68KB, %gs+4MB) for the Bary table, and [%gs+4MB, %gs+4GB) for the Tary table. MCFI always unmaps [%gs, %gs+64KB) for trapping calls to the `NULL` pointer, and uses the page [%gs+64KB, %gs+68KB) for storing trampolines, which are pointers to MCFI runtime services. Applications can be modified to jump to the trampolines to safely escape the sandbox. For example, `jmpq %gs:65536` would transfer the control to the first trampoline MCFI installs.

Figure 2.10 also shows parallel mapping of runtime-adjustable read-only data[4], 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 results from the GOT.PLT table's writability, as demonstrated by a recent attack [15]. To address this security concern, MCFI sets the GOT.PLT table to be always read-only inside the sandbox and creates outside the sandbox a shadow GOT.PLT table (by calling `shm_open`, `ftruncate`, and `mmap`), which is mapped to the same physical pages as the in-sandbox GOT.PLT table. All changes to the GOT.PLT table are therefore performed by the MCFI 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. Later in §3 and §4, we generalize the parallel mapping to support finer-grained CFGs and self-modifying code.

2.3.3 CFG Check and Update Transactions

The ID tables may be accessed concurrently by multiple threads. One thread may dynamically load a module, which triggers the generation of a new CFG. Consequently, a new set of IDs based on the new CFG needs to be put into the ID tables. At the same time, another thread may execute an indirect branch, which requires reading IDs from the tables. Since concurrent reads and writes are possible, a synchronization mechanism must be designed for maintaining consistency of the tables. Otherwise, the tables may reach some intermediate state that represents an unsound CFG and breaks program execution. A simple lock-based scheme for accessing tables

[4]Alternatively, the Bary IDs could be associated with each code module as read-only data.

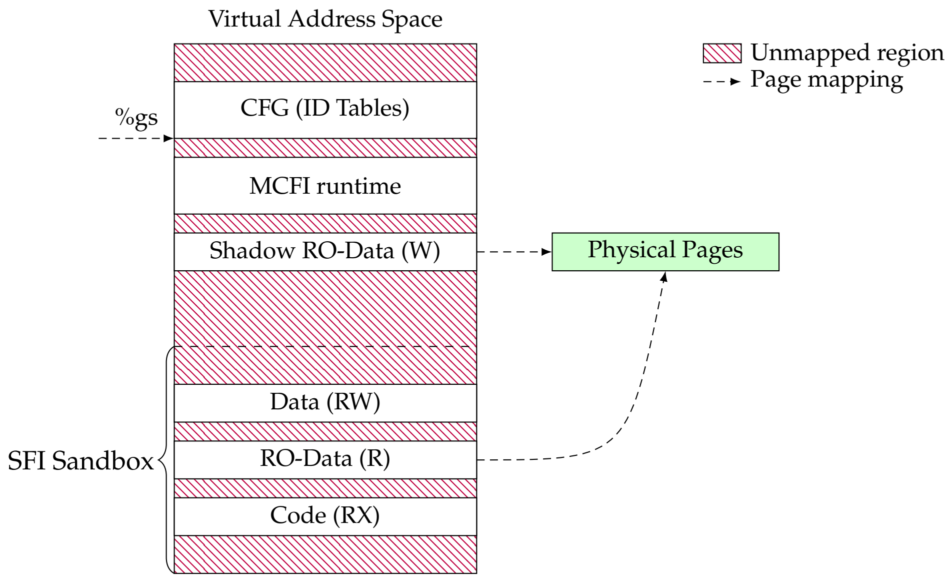


Figure 2.10: Memory layout of MCFI. "R", "W" and "X" appearing in parentheses denote the Readable, Writable, and eXecutable memory page permissions, respectively. The "RO-" prefix means Read-Only.

could be adopted, but it would incur a large performance penalty due to MCFI's table-read-dominant workloads: dynamic linking is a rare event compared to the use of an indirect branch (especially return instructions); even in Just-In-Time (JIT) compilation environments such as the Google V8 JavaScript engine, which optimizes code on-the-fly, the number of indirect branch execution is roughly 10^8 times of CFG updates triggered by dynamic code installation.

Our solution is to wrap table operations into transactions and use a custom form of Software Transactional Memory (STM) to achieve safety and efficiency. We use two kinds of transactions:

(1) *Check transaction (TxCheck).* This transaction is executed before an indirect branch. Given the address where the indirect branch is located and the address which the indirect branch targets, the transaction reads the branch ID and the target ID from the tables, compares the two IDs, and takes actions if the IDs do not match. This transaction performs only table reads.

(2) *Update transaction (TxUpdate).* This transaction is executed during dynamic linking. Given the new IDs generated from the new CFG after linking a library, this transaction updates the Bary and Tary tables.

The reason why a transaction-based approach is more efficient is that the check transaction performs speculative table reads, assuming there are no other threads performing concurrent writes; if the assumption is wrong, it aborts and retries. This technique matches our context well and provides needed efficiency.

MCFI could adopt standard STM algorithms to implement the transactions. However, those algorithms are generic and separate metadata (e.g., the version numbers) from real data (the ECNs). As a result, they require multiple instructions for retrieving metadata and real data, and multiple instructions for comparing metadata and real data to check for transaction failure and CFI violation. We micro-benchmarked the TML [38] algorithm, a state-of-the-art sequence-lock-based STM algorithm particularly optimized for read-dominant workloads, and found it is one-time slower than MCFI's custom transaction algorithm, which puts metadata and real data in a single word. The compact representation enables MCFI to use a single instruction to retrieve both meta and real data and a single instruction to check for transaction failure.

Update Transactions

When a library is dynamically linked, MCFI produces a new CFG for the program after linking. Based on the new CFG, a new set of Equivalence Class Numbers (ECNs) is assigned to equivalence classes induced by the new CFG. In the rest of this section, we assume the existence of two functions that return the new ECNs: (1) `getBaryECN` takes a code address as input and, if there is an indirect branch at that address, returns the branch ECN of the indirect branch; it returns a negative number if there is no indirect branch at the address; (2) `getTaryECN` takes a code address as input and, if the address is a possible indirect-branch target, returns the address's ECN (i.e., the ECN of the equivalence class that the address belongs to); it returns a negative number if the code is not a possible indirect-branch target.

Figure 2.11 presents the pseudocode that implements update transactions. It is implemented inside MCFI's runtime and is used by MCFI's dynamic linker to update the ID tables. An update transaction starts by acquiring a global update lock and incrementing a global version number. The lock is to serialize update transactions among threads. This simple design takes advantage of the fact that update transactions are rare in practice and allowing concurrency among update transactions does not gain much efficiency. We note that the global update lock does not prevent concurrency between update transactions and check transactions.

The update transaction performs table updates in two steps: first update the Tary table, and then the Bary table. The separation of the two steps is achieved by a memory write barrier at line 5, which guarantees that all memory writes to Tary finish before any memory write to Bary. Bary and Tary table updates cannot be interleaved; otherwise, at some intermediate state in an update transaction, some IDs in the Tary and Bary tables would have the old version and some IDs would have the new version. Consequently, check transactions would use different versions of CFGs for different indirect branches, therefore seeing an unsound intermediate CFG. By updating one table first before updating the other, check transactions either see the old sound CFG or the new sound CFG for all indirect branches at all times.

Function `updTaryTable` first constructs a new Tary table (line 11). Constants `CodeBase` and `CodeLimit` are the code region base and limit, respectively. The table construction process iterates each eight-byte aligned code address, invokes `getTaryECN`, and updates the appropriate entry in the table. The auxiliary function `setECNAndVer` updates the table entry with the ECN and the global version number; its code is omitted for brevity. After construction, the new Tary table is copied to the Tary table region with the base address in `TaryTableBase` (line 21). The `copyTaryTable` implementation is critical to the performance of update transactions. An insight is that table entries can be updated in parallel; the only requirement is that each ID update should be atomic. Therefore, we could use the weak order memory write instruction `movnti`, which directly writes data into memory without polluting the cache, to perform fast parallel copying.

Function `updBaryTable` performs similar updates on the Bary table with the help of `getBaryECN`; its pseudocode is omitted.

Check Transactions

Check transactions run during the execution of indirect branches. For efficiency, MCFI implements a check transaction as a sequence of machine instructions and instruments an indirect branch to inline the sequence. The sequence is slightly different for each kind of indirect branches

```
void TxUpdate () {
  acquire(updLock);
  globalVersion = globalVersion + 1;
  updTaryTable();
  sfence;
  updBaryTable();
  release(updLock);
}
void updTaryTable() {
  // allocate a table and init to zero
  allocateAndInit(newTbl);
  for (addr=CodeBase;addr<CodeLimit;addr+=8) {
    ecn=getTaryECN(addr);
    if (ecn >= 0) {
      entry=(addr - CodeBase) / 8;
      newTbl[entry]=0x1; // init reserved bits
      setECNAndVer(newTbl, entry,
                   ecn, globalVersion);
    }
  }
  copyTaryTable(newTbl, TaryTableBase);
  free(newTbl);
}
```

Figure 2.11: Pseudocode for implementing update transactions.

(i.e., returns, indirect jumps, and indirect calls). Further, it needs adaptation for different CPU architectures. We present the x64 sequence in this dissertation. The implementation on other CPU architectures is similar and thus omitted for brevity.

Figure 2.12 presents how a check transaction is implemented in assembly for return instructions on x64. A return instruction is translated into a `popq`/`jmpq` sequence (lines 2 and 9); this is to prevent a concurrent attacker from modifying the return address on the stack after checking. Instruction at line 3 operates on lower four bytes of `%rcx` and has the side effect of clearing the upper 32 bits of `%rcx`. As discussed, the sandbox is in the region of [0, 4GB); so the instruction at line 3 restricts the return address to be within the sandbox. Instruction at line 5 reads the branch ID from a constant index in the Bary table. Instruction at line 6 reads the target ID from the Tary table. As discussed before, both the Bary and Tary tables start from `%gs`.

Based on the values of the branch and target IDs, the following four cases may occur:

```
1    TxCheck {
2      popq  %rcx                 # pop the return address into rcx
3      movl  %ecx, %ecx           # clear the upper 32 bits of rcx
4    Try:
5      movq  %gs:ConstBaryIndex, %rdi # retrieve the Bary ID
6      movq  %gs:(%rcx), %rsi     # retrieve the Tary ID
7      cmpq  %rdi, %rsi           # compare the IDs
8      jne   Check                # if ne, IDs are not equal
9      jmpq  *%rcx                # perform the indirect jmp
10   Check:
11     testb $1, %sil             # test whether the Tary ID is valid
12     jz    Halt                 # if z, not valid
13     cmpl  %edi, %esi           # compare the versions
14     jne   Try                  # if ne, abort the check and retry
15   Halt:
16     hlt                        # CFI violation
17   }
```

Figure 2.12: Implementation of check transactions for x64 return instructions.

(1) If the branch ID in `%rdi` equals the target ID in `%rsi`, instructions at lines 7, 8 and 9 get executed, performing the control transfer. In this case, the target-ID-validity check, the version check, and the ECN check are completed by a single comparison instruction, making this common case efficient. It should be noted that the same checks might be bypassed if the attackers redirect the target to a region in [0, 4MB) where the trampoline pointers and Bary table IDs are stored, discussed in §2.3.2. However, since the region is always unmapped, the program will be trapped after the indirect control transfer.

(2) If the target address is not 8-byte aligned or its corresponding Tary ID contains all zeros, then the target ID in `%rsi` is invalid. Since the branch ID is always valid, the ID comparison fails. As a result, instructions at lines 7, 8, 11, 12, and 16 get executed and the program is terminated. In "`testb $1, %sil`", `%sil` is the lowest byte in `%rsi` and the instruction tests whether the lowest bit in `%sil` is one. If it is not one, we have a violation of the CFI policy because it uses a return address that cannot be a possible target.

(3) If the target ID is valid, but the branch ID in `%rdi` has a different version from the target ID in `%rsi`, instructions at lines 7, 8, 11, 12, 13, and 14 get executed, causing a retry of the

transaction. This case happens when an update transaction is running in parallel. The check transaction has to wait for the update transaction to finish updating the relevant IDs.

(4) If the target ID is valid, and the versions of the two IDs are the same, but they have different ECNs, instructions at lines 7, 8, 11, 12, 13, 14, and 16 get executed and the program is terminated. This case violates the CFI policy.

Indirect calls and jumps can be instrumented similarly with minor adjustments mainly for scratch registers. It is also straightforward to port the above implementation to other CPU architectures.

Linearizability. The two ID tables can be viewed as a concurrent data structure with two operations (check and update operations). One widely adopted correctness criterion in the literature of concurrent data structures is *linearizability* [39], meaning that a concurrent history of operations should be equivalent to a sequential history that preserves the partial order of operations induced by the concurrent history. Our ID tables are linearizable. In `TxUpdate`, the linearization point is right after the memory barrier at line 5. Before the point, `TxChecks` respect the old CFG; after the point, `TxChecks` respect the new CFG. In `TxCheck`, the linearization point is the target ID read instruction at line 6 when the valid target ID has the same version as the branch ID or the target ID is invalid.

ABA problem. MCFI's ID-encoding scheme supports 2^{28} versions and it might encounter the ABA problem [40]. For example, an attacker may load over 2^{28} modules and exhaust the MCFI's version number space. This is unlikely in practice, even for just-in-time compiled code. Security is violated only if the program has at least 2^{28} code updates during a check transaction. To avoid this issue, MCFI maintains a counter of executed update transactions and makes sure it does not hit 2^{28}. After completion of an update transaction, if every thread is observed to have finished using old-version IDs (when each thread invokes a system call or runtime trampoline calls), the counter is reset to zero.

Dynamic code unloading. In addition to dynamic code loading, MCFI supports dynamic library unloading. When a library is unloaded, all indirect branch targets inside the library's code are marked invalid, achieved by changing the validity bits of IDs in the Tary table to all zeroes.

This prevents all threads from entering the library's code, since there should be no direct branches targeting the library. However, there might be threads currently running or sleeping in the library's code. Hence, it is unsafe to reclaim 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, MCFI asynchronously waits until it observes that all threads have executed at least one system call or runtime trampoline call; we instrument 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 after every counter has been incremented.

MCFI transactions versus Sequence Locks

MCFI transactions are similar to sequence locks such as Transactional Mutex Locking (TML) [38], which defines a global sequence number `S` (like MCFI's version) starting from an even number (usually 0). When TML-protected data structures, such as Bary and Tary, are read, three steps are conducted: (1) the global sequence number is read and kept in a local variable `v`; if `v` is odd, then the data structures are being concurrently updated, the program re-executes step 1; (2) the data structures are read; (3) the global sequence number is read again and compared with the previously read one. If `v == S`, it indicates that during the read operation, the data structures were not altered, so the program has read a consistent snapshot; otherwise the three steps are re-executed. When TML-protected data structures are changed, the writer first increments `S` by one to make it odd, then makes changes and finally increments `S` by one to make it even again.

MCFI's check transaction implementation has two advantages over TML: (1) TML requires four memory reads to read the versions and IDs, while MCFI needs only two, therefore TML consumes twice as much time as MCFI; (2) when implemented on other CPU architectures that may reorder memory reads, load fences need to be added between each step in TML, which harms the performance. However, MCFI's two memory read instructions do not need to be serialized, since their execution order does not matter.

In-place Update versus Copy-On-Write Update

The MCFI update transaction performs in-place updates, but Copy-On-Write (COW) update schemes could alternatively be used, which replace the old tables with newly allocated ones. For example, since both Bary and Tary tables are indexed by the `%gs` segment register, we could change the register to point to new Bary and Tary tables after CFG generation, and delete the old tables if all threads have finished referencing the old tables (e.g., when all threads have been observed to execute system calls or trampoline calls at least once, similar to how we deal with the ABA problem and dynamic code unloading previously.)

However, the COW schemes in general may use an unbounded amount of memory. For instance, if a thread has a lower priority than other threads and rarely gets a time quantum to run, loading any new module will allocate new tables but not free the old tables, which might exhaust physical memory. Fortunately, the in-place update avoids this issue.

2.4 Interoperability

To be practical, MCFI should be interoperable, meaning that code instrumented with MCFI can be linked with uninstrumented modules (could also be JITted code) without breaking the program. Interoperability allows incremental development and deployment of software modules, which are critical to software engineering. However, the check transactions described in Figure 2.12 do not support interoperability and need to be adjusted. Otherwise without adjustments, the program will break, since uninstrumented modules do not have CFG-metadata associated and sound ID assignment is generally infeasible. As a result, we reserve the byte `0xfc` for interoperability use, and guarantee that any Bary and Tary ID should not contain any byte equal to `0xfc` (this design slightly reduces the maximum number of supported equivalence classes and versions). When an uninstrumented module is loaded, we populate all its code's corresponding Tary bytes using `0xfc`. The check transactions are accordingly changed to what Figure 2.14 shows. Compared to Figure 2.12, we add two extra instructions at line 12 and 13 that check whether an instrumented indirect branch is jumping into an uninstrumented module. If so, the control is transferred to the target; otherwise the target is in an instrumented module and therefore the

```
1 /* An MCFI-instrumented
2    module A */
3 typedef void (*F)(char*);
4 typedef void (*G)(long);
5 void fx(char*) {}
6 void fy(G fp)  {
7  /* False CFI violation */
8   fp(0);
9 }
10 ...
11 /* function fz is defined
12    in module B */
13 void fz(F);
14 ...
15 /* the following calls into
16    module B */
17 fz(fx);
```

```
1 /* An uninstrumented
2    module B */
3 void fz(F fp) {
4   /* calls back to module A */
5   fy((G)fp);
6 }
```

Figure 2.13: A compatibility condition violation introduced by the interoperability support.

same checks are performed.

Another point worth mentioning is that when an uninstrumented module is linked, the compatibility condition (§2.2.2) for the sound CFG generation may not hold any more. Figure 2.13 shows such an example of two modules: module A (on the left) is MCFI-instrumented and module B (on the right) is uninstrumented. Module A calls module B at line 17 and passes the address of function `fx` of type `F` to module B. Then, module B casts `fx`'s address to type `G` and passes it back to module A, so the function pointer `fp` at line 7 will hold the address of `fx`. However, since `fp` has a different type from `fx`, the dereference at line 8 will be falsely detected as a CFI violation. Consequently, we conservatively merge all functions whose addresses are taken into one equivalence class to tolerate possible compatibility condition violations. In addition, all return addresses are merged into another equivalence class. Essentially, coarse-grained CFI is enforced for interoperability.

The above described solution works for uninstrumented modules that do not invoke system calls directly, which should be the common case. However, for those modules that directly invoke system calls, our current implementation might break code, because of the SFI isolation. For example, if an uninstrumented module allocates a memory page outside of the sandbox (by issuing

```
TxCheck {
  popq  %rcx                # pop the return address into rcx
  movl  %ecx, %ecx          # clear the upper 32 bits of rcx
Try:
  movq  %gs:ConstBaryIndex, %rdi # retrieve the Bary ID
  movq  %gs:(%rcx), %rsi    # retrieve the Tary ID
  cmpq  %rdi, %rsi          # compare the IDs
  jne   Check               # if ne, IDs are not equal
Go:
  jmpq  *%rcx               # perform the indirect jmp
Check:
  cmpb  $0xfc, %sil     # test if the target is uninstrumented
  je    Go                  # if e, jump to the target
  testb $1, %sil            # test whether the Tary ID is valid
  jz    Halt                # if z, not valid
  cmpl  %edi, %esi          # compare the versions
  jne   Try                 # if ne, abort the check and retry
Halt:
  hlt                       # CFI violation
}
```

Figure 2.14: Interoperable check transactions for x64 return instructions.

the mmap system call) and passes a page pointer to an MCFI module for write. Since the MCFI module cannot directly modify the page outside of the sandbox, the program will be trapped. Therefore, to achieve general interoperability, the MCFI runtime needs to be moved into the OS kernel, similar to how Control-Flow Guard is implemented in Windows 10. With such an implementation, we could cut the virtual address space into two halves: the lower half [0, 64TB) is used for application code and data, while the upper half [64TB, 128TB) is for Bary and Tary tables. The tables are always read-only in the user-space, thus removing the need for SFI. If they need to be updated, the kernel remaps them to writable pages in the kernel space and changes their contents. In addition, the current runtime services should be redesigned as system calls.

2.5 Implementation

The MCFI toolchain basically has two tools: an LLVM-based C/C++ compiler, which performs code instrumentation and generation of CFG-related metadata; and a runtime that loads instrumented modules and monitors their execution.

The MCFI compiler is modified from Clang/LLVM-3.5, with a `diff` result of about 4,500 lines of changes. In summary, the changes to LLVM propagate metadata such as class hierarchies and type information for generating the CFG. The metadata are inserted into the compiled ELF as new sections. In addition, each MCFI-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`. 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 is changed to allocate a memory region inside the sandbox and execute `sigaltstack` to switch the stack to the in-sandbox region, which is released when the thread exits. Security analysis of this design is presented in §5.1.6.

The MCFI runtime consists of around 11,000 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 accordingly. The CFG is generated using the metadata in the code modules.

2.6 Evaluation

We evaluated MCFI on a PC running x64 Unbuntu 14.04.3. The computer has an Intel Xeon E3-1245 v3 processor and 16GB memory. We chose SPECCPU2006 C/C++ benchmark programs to measure the CFG precision and performance, and compiled all programs with the O3 optimization level and stack cookie protection off.

2.6.1 CFG Statistics

Equivalence Classes

MCFI supports program-dependent numbers of equivalence classes. Table 2.4 shows the numbers. The “IBs” column lists the total number of instrumented indirect branches, with the number of those indirect branches that have targets shown in the parentheses. For example, a libc function `aio_cancel` is never called by any of the benchmarks, so its return instruction has nowhere to target. The “IBTs” column presents the number of all indirect branch targets; the “EQCs” column presents the number of equivalence classes of the indirect branch targets. Moreover, the “Avg IBTs / IB” lists how many targets an indirect branch has on average, and the “Avg IBs / IBT” shows the number of indirect branches that could reach the same target on average. As can be seen, MCFI indeed supports fine-grained CFGs. The average targets per indirect branch and average indirect branches per target are much less than coarse-grained CFI, which could be as many as the number of indirect branch targets and indirect branches, respectively. Further, indirect branches can reach more targets in some programs (e.g., `403.gcc`) than others, and it is because those programs have function pointers that could indirectly reach many functions or functions that are called in many places. For example, `403.gcc` defines different functions for emitting different instructions, and these functions are all indirectly callable and share the same type signature.

Distribution of Edges

In addition to the average results shown before, we calculated the edge distribution, and list the detailed results in Table 2.5. Each cell shows how many indirect branches (in percentage) have the number of targets specified in the cell’s column header. For example, 47.4% of indirect branches in `400.perlbench` have less than ten targets. As can be seen from the table, on average about 66.1% indirect branches have less than ten targets, and nearly 86.7% indirect branches have less than a hundred targets. The other 13.3% indirect branches have no less than 100 targets, and 7.9% even has one thousand targets or more. These indirect branches are either indirect calls that can reach thousands of functions (e.g., `445.gobmk`) or return instructions whose functions have many call sites.

SPECCPU2006	IBs (with matching targets)	IBTs	EQCs	Avg IBTs / IB	Avg IBs / IBT
400.perlbench	3327 (2399)	18379	1039	722	95
401.bzip2	1711 (943)	4065	505	33	8
403.gcc	6108 (5039)	50413	2321	1244	125
429.mcf	1625 (875)	3852	493	34	8
433.milc	1825 (1030)	5880	625	36	7
444.namd	4796 (3042)	17620	1314	154	27
445.gobmk	3908 (3119)	14557	944	949	204
447.dealII	13624 (8361)	61464	3225	1035	141
450.soplex	6305 (4407)	22418	1847	175	35
453.povray	6275 (4355)	28738	2027	374	57
456.hmmer	2038 (1136)	7907	682	93	14
458.sjeng	1777 (1010)	4827	560	32	7
462.libquantum	1688 (917)	4170	514	35	8
464.h264ref	2455 (1616)	7047	793	41	10
470.lbm	1612 (867)	3840	485	35	8
471.omnetpp	7791 (5526)	35772	2203	456	71
473.astar	4770 (2994)	16763	1325	159	29
482.sphinx3	1893 (1071)	6432	652	39	7
483.xalancbmk	31167 (27117)	97265	7970	1103	308

Table 2.4: CFG statistics for SPECCPU2006 C/C++ benchmarks.

SPECCPU2006	$[1, 10)$	$[10, 100)$	$[100, 1000)$	$[1000, +\infty)$
400.perlbench	47.4%	23.9%	6.7%	22.0%
401.bzip2	68.6%	24.3%	7.1%	0%
403.gcc	56.4%	21.7%	4.7%	17.2%
429.mcf	67.0%	25.8%	7.2%	0%
433.milc	66.9%	25.1%	8.0%	0%
444.namd	73.3%	18.6%	2.3%	5.8%
445.gobmk	32.1%	10.8%	6.2%	50.9%
447.dealII	72.9%	15.1%	4.1%	7.9%
450.soplex	77.6%	14.6%	2.8%	5.0%
453.povray	71.1%	17.0%	2.0%	9.9%
456.hmmer	67.4%	23.9%	2.2%	6.5%
458.sjeng	65.5%	28.2%	6.3%	0%
462.libquantum	66.2%	26.0%	7.8%	0%
464.h264ref	75.2%	17.6%	7.2%	0%
470.lbm	66.1%	26.6%	7.3%	0%
471.omnetpp	68.1%	14.9%	7.6%	9.4%
473.astar	74.8%	17.4%	1.9%	5.9%
482.sphinx3	68.8%	23.0%	8.2%	0%
483.xalancbmk	71.4%	17.2%	2.0%	9.4%
Avg	66.1%	20.6%	5.4%	7.9%

Table 2.5: Edge distribution for SPECCPU2006 C/C++ benchmarks.

CFG Precision Loss

In addition, we measured the CFG precision loss mentioned in §2.2.3. For the loss induced by tail-call elimination, we disabled the optimization at the machine code level and collected the

SPECCPU2006	IBs (with matching targets)	IBTs	EQCs	Avg IBTs / IB	Avg IBs / IBT
400.perlbench	4036 (2747)	19362	1469	18	3
401.bzip2	2155 (1100)	4572	727	9	3
403.gcc	7362 (5865)	52754	3567	70	8
429.mcf	2067 (1034)	4353	710	9	3
433.milc	2289 (1196)	6422	868	10	2
444.namd	5669 (3496)	18603	1719	37	7
445.gobmk	4667 (3566)	15453	1310	19	5
447.dealII	15963 (10177)	64095	4121	222	36
450.soplex	7308 (4934)	23591	2365	32	7
453.povray	7331 (4978)	30069	2606	41	7
456.hmmer	2547 (1319)	8505	956	11	2
458.sjeng	2234 (1181)	5348	794	10	3
462.libquantum	2142 (1085)	4555	756	9	3
464.h264ref	2949 (1814)	7652	1085	8	2
470.lbm	2059 (1029)	4346	705	9	3
471.omnetpp	8820 (6129)	36957	2703	95	16
473.astar	5650 (3460)	17748	1736	37	8
482.sphinx3	2373 (1253)	6994	916	10	2
483.xalancbmk	33933 (29142)	100641	9321	133	39

Table 2.6: CFG statistics for SPECCPU2006 C/C++ benchmarks without tail-call elimination.

CFG statistics in Table 2.6. Compared to Table 2.4, the average IBTs per IB and average IBs per IBT decrease by 84.5% and 81.5%, respectively, indicating relatively large precision recovery with the optimization turned off. As a result, it is useful to further study how to selectively turn on/off the optimization for specific returns to meet performance and security goals, although our experimental results show that even completely disabling the optimization incurs only 0.1% more performance overhead than enabling the optimization.

Besides, we measured the CFG precision loss caused by equivalence class partitioning (with the tail-call elimination turned off), which is also described in §2.2.3 as three cases. For the case of return address merging due to indirect calls, we calculated the number of different return address sets without equivalence class partitioning, denoted by N, and the number of equivalence classes of return addresses, represented by M, and report the loss ratio defined as $1 - M/N$ in Table 2.7. On average, 12.2% of return address sets are merged into equivalence classes. It is worth mentioning that different programs report different loss ratios. Basically, the more functions that can be indirectly called, the more precision is lost. For example, C++ programs tend to have larger loss ratios (avg. 28.8%) than C programs (7.4%) because many functions are virtual methods that

SPECCPU2006	Equivalence class loss ratio
400.perlbench	22.1%
401.bzip2	5.0%
403.gcc	19.6%
429.mcf	4.9%
433.milc	4.4%
444.namd	19.6%
445.gobmk	45.0%
447.dealII	38.6%
450.soplex	28.8%
453.povray	22.5%
456.hmmer	4.8%
458.sjeng	5.0%
462.libquantum	4.7%
464.h264ref	5.2%
470.lbm	4.9%
471.omnetpp	32.5%
473.astar	18.6%
482.sphinx3	4.4%
483.xalancbmk	55.3%
Geomean	12.2%
Geomean (C)	7.4%
Geomean (C++)	28.8%

Table 2.7: Equivalence class loss due to indirect call-triggered equivalence class merging of returns.

Program	Equivalence class loss ratio
444.namd	27.8%
447.dealII	31.5%
450.soplex	25.4%
453.povray	23.6%
471.omnetpp	21.8%
473.astar	26.7%
483.xalancbmk	34.3%
Geomean	27.0%

Table 2.8: Equivalence class loss in the case when C++ virtual method calls are legally allowed to invoke virtual methods defined in super classes.

are indirectly reachable.

Similarly, we calculated the equivalence class loss ratios caused by allowing virtual method calls to target methods in super classes, and report the results in Table 2.8. On average, the loss ratio is about 27%.

Finally, Table 2.9 summarizes the loss caused by C++ method pointers. The "Unique Method

SPECCPU2006	Unique Method Pointer Dereferences	# of Merged Member Method EQCs
447.dealII	1	1
483.xalancbmk	7	20

Table 2.9: Equivalence class loss due to C++ method pointers.

Pointer Dereferences" column shows how many different types of method pointers are dereferenced in the final binary, and the "# of Merged Method EQCs" presents how many different member method equivalence classes are merged because of being pointed to by the same type of method pointer. Only two C++ benchmarks, `447.dealII` and `483.xalancbmk` use method pointers. However, `447.dealII` does not lose any precision since only one method matches the type of the method pointer; `483.xalancbmk` loses some precision, because the method pointers merge a few more different member method equivalence classes.

CFG Generation Time

We measured the CFG generation time for each benchmark, and observed the maximum time on `483.xalancbmk`, which is about 0.5 second.

The MCFI CFG generation consists of four phases: metadata processing, edge connection, equivalence class partitioning and ID filling. The metadata processing combines metadata of all modules and linearly parses the metadata to construct class hierarchies and represent virtual method calls, function pointers, returns, functions and return addresses as graph nodes. The edge connection phase simply connects indirect branches to their targets using *undirected* edges according to §2.2.1. In the resulted graph, each connected component is an equivalence class, and a Breadth-First Search (BFS) procedure is then repeatedly performed on the graph to extract all connected components, so the time complexity of this phase is $O(V + E)$, where V is the number of nodes and E denotes the number of edges. The ID filling phase, which conducts table update transactions that assign each equivalence class an ID and copy the ID to corresponding Bary and Tary entries, is also a linear procedure.

Each of the four phases costs different periods of time, and Table 2.10 lists the average time as a percentage compared to the total CFG generation time. As can be seen, the metadata parsing consumes more than half of the CFG generation time. If the CFG generation time is considered

Metadata processing	55.3%
Edge connection	22.8%
Equivalence class partitioning	12.5%
ID assignment and filling	9.4%

Table 2.10: CFG generation time decomposition.

intolerable for larger programs, the metadata processing could be optimized (e.g., using a more compact encoding for the metadata).

2.6.2 Performance Evaluation

MCFI inserts checks into programs and programs protected with MCFI is slower and larger. We present our measurements on SPECCPU2006 benchmarks.

Execution Time Overhead

We ran SPECCPU2006 benchmarks over the reference data sets for three times and calculated the average running time (with the variance less than 1%). Then, we compared the running time of MCFI-protected programs (including the CFG generation time) with that of native programs and calculated the overhead, depicted as percentages in Figure 2.15. On average, MCFI slows down program execution by 2.9%.

Two points are worth mentioning. First, notice that several benchmarks (e.g., 450.soplex) run even faster with MCFI's instrumentation. We replaced the MCFI instrumentation with nops and still observed the acceleration (e.g., 0.6% faster for 450.soplex), therefore we believe the reason is the extra alignments MCFI requires for indirect branch targets. Second, MCFI incurs different overhead over different programs, and it is positively correlated with the execution frequency of indirect branches. We calculated the correlation using the Pearson correlation coefficient and got the result of 0.74, which indicates strong correlation.

To demonstrate interoperability, we linked instrumented SPEC C++ programs with uninstrumented `libc++` and `libc++abi` and successfully tested their execution on the same reference data sets. The performance overhead is similar to the above runs with all libraries instrumented.

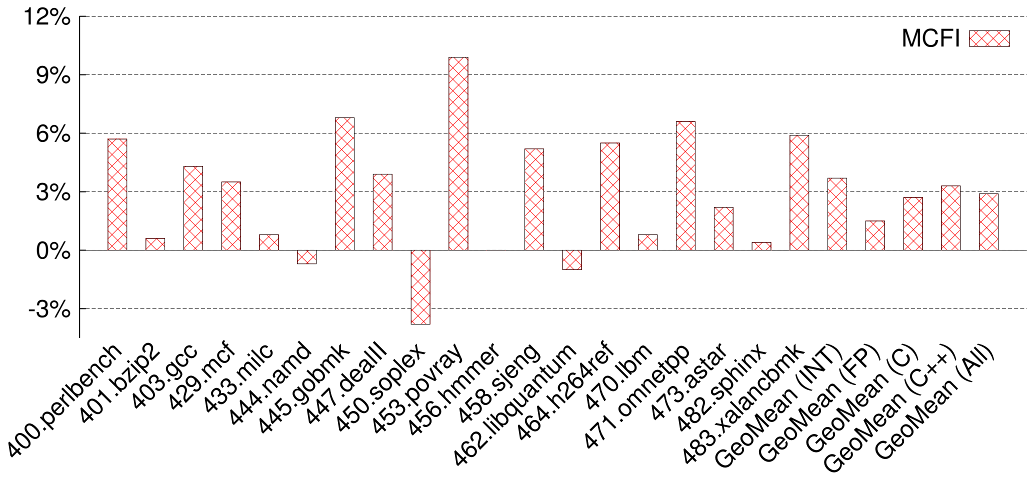


Figure 2.15: MCFI runtime overhead on SPECCPU2006 C/C++ benchmarks.

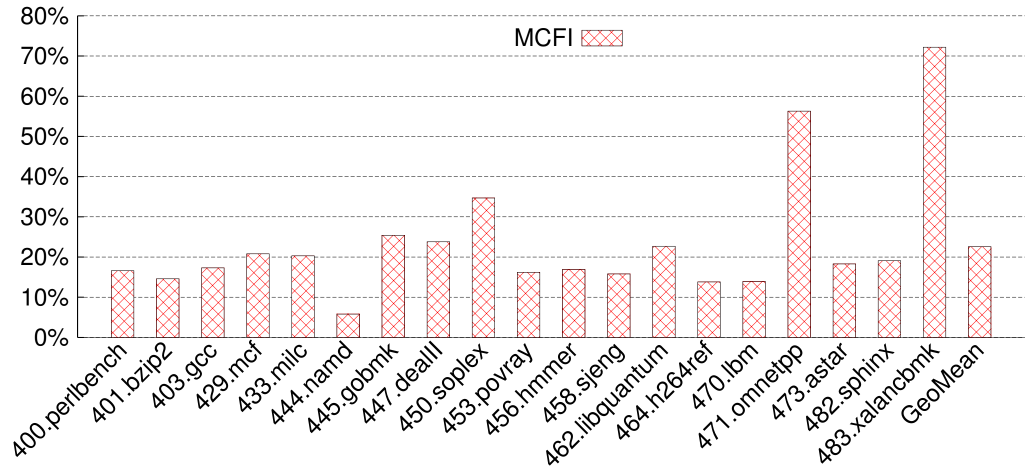


Figure 2.16: MCFI code size increase on SPECCPU2006 C/C++ benchmarks.

In addition, we tested the performance overhead when disabling the tail-call elimination optimization at the machine-code level, which was about 3%. Considering the negligible performance slowdown and the relatively big precision recovery, it might be beneficial to generally turn the optimization off, especially for non-compute-intensive modules.

Space Overhead

Figure 2.16 shows the code size increase incurred by MCFI on SPECCPU2006 benchmarks. On average, the code bloat is around 22.6%, due to MCFI's checks and alignments. C++ programs tend to have a larger bloat after MCFI's instrumentation because of higher density of indirect branches and indirect branch targets. At runtime, the Bary and Tary tables occupy nearly the same amount of memory as the code, so compared to the native counterparts, MCFI needs extra memory of around 1.2 times of the code size. However, the memory footprint increase is negligible ($< 0.8\%$), because the most of the runtime memory consumption is about data.

2.7 Future Work

Condition violation checker. MCFI generates sound CFGs for only those C/C++ programs that satisfy the compatibility condition in §2.2.2. Our current checker captures bad type casts and rules out those "safe" bad casts. However, there are still quite some violations left for manual investigation. We could, in the future, add data-flow analyses to the checker to filter out more false positives.

CFG loss reduction. As mentioned, three cases of CFG precision loss exist because the current instrumentation design requires equivalence class partitioning. Therefore, it would be interesting and beneficial to further improve (or redesign) the instrumentation to reduce the CFG precision loss.

Portability. MCFI is currently only implemented on x64, and it would be beneficial to port it to other CPU architectures such as ARM and POWER. Since the transactions of MCFI only require that word-aligned read and write instructions are atomic, which are supported by most CPUs, porting of the transaction implementation should be straightforward. The SFI sandbox implementation may need some changes as other CPUs might not support address overriding, but SFI as a general approach for implementing user-level isolation may be implemented in many other ways. Alternatively, the runtime can be implemented inside the OS kernel so that the SFI sandbox is no longer needed. In addition, it would be interesting to implement MCFI in other mainstream OSes such as Windows and OSX.

Interoperability. MCFI currently supports interoperability between instrumented modules and uninstrumented modules that do not directly invoke system calls as previously discussed in §2.4. To support uninstrumented modules that may directly issue system calls, the runtime needs to be implemented as part of the OS kernel to eliminate the SFI isolation. In addition, our current implementation merges all equivalence classes of each kind of indirect branches when an uninstrumented module is loaded, which deteriorates the protection. Hence, another possible research direction about interoperability is how to still enforce fine-grained CFI for instrumented modules. For instance, if none of an uninstrumented module's exported functions accepts function pointers as arguments, it is probably safe not to merge equivalence classes for functions, since there exists no data flow (except through other media such as files) that could pass a function address in an instrumented module to an instrumented indirect call with an unmatched type, assuming the compatibility condition is met in all instrumented modules. Binary analysis can also be conducted to investigate any binary module to see if the compatibility condition might be violated in the uninstrumented module. Further, if there exists no indirect tail-jumps and direct tail-jumps to PLT entries, the equivalence classes for returns may neither need to be merged, since there exists no tail-call chain that would connect an instrumented return to a return address in an instrumented module.

Backward compatibility. Backward compatibility means that an MCFI-instrumented code module can run on any existing systems that are unaware of MCFI at all. For example, MCFI-instrumented SPEC benchmark binaries should run on current Ubuntu without any system patches. This is a nice-to-have feature for practicality since it eases software maintenance and distribution. Apparently our current implementation of the check transactions does not support this feature, because the `%gs` register and the ID tables will are not set by current systems. However, a feasible solution would be that after MCFI instrumentation the compilation toolchain replaces the instrumentation with nops but remembers the instrumentation code bytes as metadata. For example, the `0x67` prefix added for memory write sandboxing could be replaced with `0x90`, which is a `nop`. In addition, special flags need to be added to the generated ELF file to indicate MCFI. As a result, existing systems that are unaware of the special flags would execute the patched code, while MCFI-aware systems would patch the nops back to the instrumentation code and execute the protected binary.

MCFI for the OS kernel. Most research about CFI focuses on application protection, but the OS kernel also needs hardening since most kernels are written in C/C++ as well. KCoFI [41] uses ClassicCFI to secure a FreeBSD kernel, so it is reasonable to believe that MCFI can protect the kernel with the support of dynamically loadable kernel modules.

2.8 Summary

This chapter presents MCFI, the first CFI approach that supports fine-grained CFGs, modularity, efficiency and interoperability. MCFI adopts a source-level semantics-based CFG generation approach to extract fine-grained CFGs. The generated CFG is then encoded as two arrays, which contains specially designed IDs to enable efficient table look-ups and updates that are implemented as lightweight STM transactions. Moreover, the ID encoding and instrumentation support interoperability.

The content of this chapter is based on our previously published papers [27] and [28] (the C++ CFG generation part). The major differences between this chapter and the published papers are the followings:

- This chapter presents the 8-byte ID encoding scheme on x64, while the published papers discuss a 4-byte ID encoding scheme on both x64 and x86. As a result, the ID-encoding-dependent Bary and Tary table query and update operations differ as well. On x64, 8-byte IDs should provide better practicality than 4-byte IDs due to the much larger equivalence class amount and version space.

- The memory write sandboxing described in this chapter uses a hardware address-override prefix [37], while the published papers use the same technique as MIP [12], which needs more instrumentation and is slower.

- This chapter details the general process of dynamic code unloading, which is not mentioned in the published papers.

- This chapter allows a variadic function pointer to reach only those functions with literally the same type. However, the published papers allow variadic function pointers to also target non-variadic functions with the matching return type and known parameter types. For example, a function pointer `fp` of type `void (*)(int, ...)` can reach only functions of type `void (*)(int, ...)` in this chapter, but the same function pointer can target functions of type `void (*)(int, int)` in the published papers besides those functions of type `void (*)(int, ...)`. Although allowing variadic function pointers to target non-variadic functions could reduce code retrofitting effort due to less bad type casts, the CFG precision is lower.

- This chapter discusses a new protection scheme for GOT.PLT, which does not need instrumentation for PLT entries. However, PLT entry instrumentation is needed in the published papers.
- This chapter presents new experimental data such as CFG precision loss and CFG generation time. The libraries are dynamically linked in this chapter (and others), but statically linked in the published papers. The CPU and OS used in measurements are also different.

Chapter 3

Per-Input Control-Flow Integrity

3.1 Overview

In this chapter, we consider improving the precision of CFGs generated by MCFI. Although we might adopt other static CFG generation algorithms, the biggest concern is that for any program there exists a minimal sound CFG that cannot be further improved by any static analysis. The undecidability of generally generating such a minimal sound CFG makes things even worse (see §1.3.3 for details). However, notice that statically generated CFGs contain edges for *all* inputs, there might still exist many redundant edges for a given concrete input. Therefore, for each input, we may trim those unnecessary edges from the static CFG to build a more precise CFG. Essentially, we describe how to generate *per-input* CFGs and enforce such CFGs in a new CFI scheme called Per-Input CFI (PICFI or πCFI).

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

However, two challenges still remain to be addressed. The first challenge is, 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 statically generates a CFG leveraging MCFI's source-level semantics-based CFG generation. Then, πCFI starts executing 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 nops after their first execution, minimizing performance overhead.

Before proceeding, we introduce some terminology that will make the following discussion more convenient. Conceptually, a CFI method involves two kinds of CFGs:

- A static, all-input CFG. This is typically computed by static analysis. We call this CFG a *Static CFG*, abbreviated to SCFG. The CFG generated by MCFI is an SCFG.
- The CFG that is dynamically 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 MCFI and all previous CFI systems, SCFG = ECFG, and we call them *conventional CFI*. In πCFI, SCFG $\supseteq$ ECFG, since πCFI uses the SCFG to upper-bound the ECFG as described.

```
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 3.1: A motivating example for per-input CFGs.

3.1.1 Motivation for per-input CFGs

Different from conventional CFI enforcement techniques, πCFI's ECFG is computed for each specific input. We next use a toy C program listed in Figure 3.1 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 cookies in our discussion since they are orthogonal defense mechanisms to CFI.)

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

```
 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 3.2: Edge-addition instrumentation for the motivating example.

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 3.2 so that the program can add its required edges during execution. (Instead of edge addition, πCFI actually uses address activation, which will be detailed later in §3.1.2.) 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 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 3.1 can also be protected by defenses that protect the stack through a shadow stack. For instance, XFI [17] 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 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 reuses MCFI's sound SCFG generation (§2.2.1) that already handles unconventional control flow and it always adds necessary edges before they are needed by indirect branches (discussed in §3.2.2). In fact, πCFI can be built on top of any conventional CFI that generates sound SCFGs.

However, since πCFI does not perform edge removal (except during code module unloading), one worry is that its ECFG grows along with the program execution. In theory, an attacker might use some malicious input to trigger the addition of all edges 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. However, we believe πCFI offers benefits even for such programs, for the following reasons:

- An attacker would need to find a set of inputs that can trigger the addition of all edges of her/his interest; this is essentially asking the attacker to solve the code coverage problem, a traditionally hard problem in software testing.
- Our experiments 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.* [42] 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. 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 processing client 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.

3.1.2 From edge addition to address activation

The simple instrumentation shown in Figure 3.2 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 nops after its first execution. An idempotent operation is designed so that the effect of performing it arbitrary times is the same as the effect of conducting it only once. Therefore, after the first time, there is no need to perform it again. For example, the operation at line 19 in Figure 3.2 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 nops 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 target address is activated, can the address be reachable by indirect branches*. 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 multiple times has the same effect as activating it only once.

3.2 System Design

In this section, we discuss the detailed system design of πCFI, including how it achieves secure online code patching, how it activates addresses for each kind of indirect branch target addresses, and how it is made compatible with typical software features.

3.2.1 Secure code patching

Idempotent address-activation operations allow πCFI to patch the operations with nops 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

[1]The edge addition happens only once in the code of Figure 3.2, but in other examples such an operation may be executed multiple times, for instance, when it is in a loop.

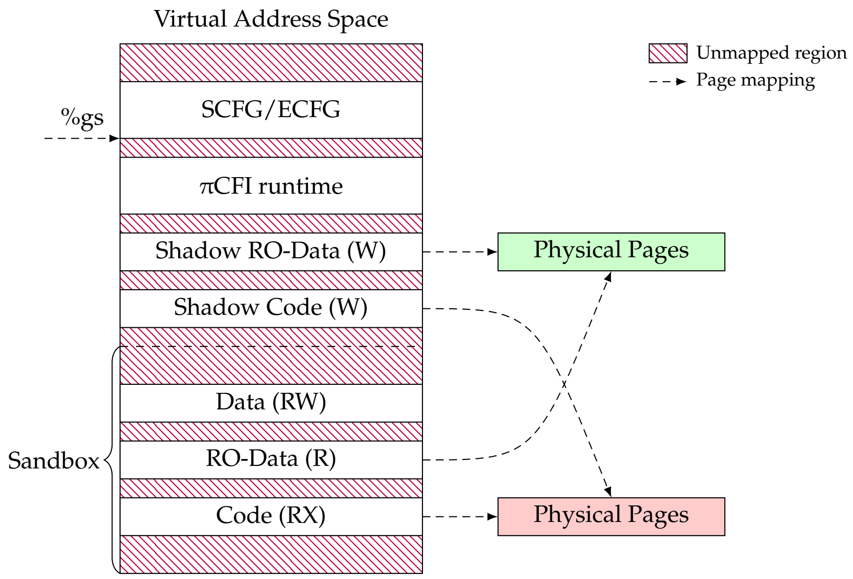


Figure 3.3: Memory layout of πCFI. "R", "W" and "X" appearing in parentheses denote the Readable, Writable, and eXecutable memory page permissions, respectively. The "RO-" prefix means Read-Only.

avoid such risks, we extend MCFI's memory layout (detailed in §2.3.2) for secure code patching.

Figure 3.3 shows the memory layout of an application protected with πCFI. The application should have been compiled and instrumented by πCFI's compilation toolchain, which is an extension of the MCFI toolchain. The application and all its instrumented libraries are loaded into the sandbox created by the πCFI runtime using the same technique as in MCFI.

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.

3.2.2 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 MCFI (or 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 nops 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, 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 nops 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.* [43]. 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 still use x64 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.

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 3.4 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 nops (line 3) to align `L` to an 8-byte aligned address. πCFI's implementation is based on MCFI, which requires all target addresses are 8-byte aligned.

2. When the code is loaded into memory by πCFI's runtime, the immediate operand of the call

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

instruction (line 4) is replaced with an immediate called `patchstub`, as shown in Figure 3.4 (b). Therefore, the call is redirected to `patchstub`, whose code is listed in Figure 3.5.

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 3.5) to `%r11`, which can be used as a scratch register thanks to the calling convention of x64 Linux. It then invokes the `return_address_activate` service provided by πCFI's runtime.

4. The runtime, once entered, saves the context and activates `L` by updating the ECFG. πCFI reuses MCFI's Tary table for encoding an ECFG. After πCFI generates the SCFG and computes all the Tary IDs, their validity bits are set to all zeroes, making the ECFG essentially empty. During address activation, the validity bits are changed to the valid encoding (see §2.3.1 for more details) by flipping the least significant bit to one.

5. The runtime next 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 3.4 (c). Finally, the runtime restores the context and jumps to line 4 in Figure 3.4 (c) to execute the patched call instruction. It should be noted that the x64 CPU automatically synchronizes the instruction cache if the new code is *jumped* to after modification [44]. On other CPU architectures (e.g., ARM), it may be necessary to use cache cleaning instructions (e.g., `DSB` and `ISB` on ARM [45]) to synchronize the instruction cache and data cache for guaranteeing the execution of the new instruction.

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

```
patchstub:
  popq %r11
  jmpq %gs:return_address_activate
```

Figure 3.5: The patch stub for activating return addresses.

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

Figure 3.6: πCFI activates a return address following an indirect-call instruction. `L` is 8-byte aligned.

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 nops 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 3.6. L is always aligned to an 8-byte aligned address by appropriately inserting nops, which are not shown. The mcfi-check at line 3 is a pseudo-operation that performs MCFI checks (detailed in §2.3.3) and can also be implemented using any conventional CFI checks. In addition, πCFI inserts a 5-byte nop (line 4) at compile-time (Figure 3.6 (a)) so that at load time a direct call to the patchstub can be inserted (Figure 3.6 (b)). Note that in this case when patchstub gets called its stack pop instruction (line 2 in Figure 3.5) 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 nop (Figure 3.6 (c)). The patch is atomic because an indirect call instruction through a register in x64 is encoded with either 2 or 3 bytes; therefore, the patched bytes will always stay within [L-8, L).

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 3.7, 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 3.8 presents part of the code that is compiled from the example in Figure 3.7 and the lea instruction at line 4 in Figure 3.8 takes the address of bar. Before the instruction, πCFI's compilation inserts a direct call to patchstub_at (at line 2 in Figure 3.8 (a)), which is another stub similar to Figure 3.5 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

```
void foo(void) {}
void bar(void) {}
void (*fp) = &foo;
int main() {
  void (*bp) = &bar;
  fp();
  bp();
}
```

Figure 3.7: Example code for function address activation.

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

Figure 3.8: πCFI's instrumentation for activating a function address.

up the symbol table again to find the address of `bar`. Appropriate nops 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 nop shown in Figure 3.8 (b).

Furthermore, C++ code can take the addresses of non-virtual methods. Such addresses are activated in the same way as a function address; that is, they are activated at the places where the addresses are taken.

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 3.9. 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 3.10 (a) shows the

```
class A {
  public:
    A() {}
    virtual void foo(void) {}
    virtual void bar(void) {}
};
class B : A {
  public:
    B() : A() {}
    virtual void foo(void) {}
};
int main() {
  B *b = new B;
  b->foo();
  A *a = new A;
  a->foo();
}
```

Figure 3.9: Example C++ code for virtual methods' address activation.

prologue of `A`'s constructor `A::A`, which is 8-byte aligned. When the code is loaded into memory, as shown in Figure 3.10 (b), πCFI's runtime changes the prologue to a direct call to `patchstub_vm` (which is another stub similar to `patchstub` in Figure 3.5 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 3.5) 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.

Moreover, since uninstrumented modules may implement C++ class constructors or statically

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

Figure 3.10: πCFI activates a virtual method by instrumenting and patching a C++ class constructor `A::A`, which is 8-byte aligned.

construct a class without calling any constructors, meaning that class objects' virtual methods may not be appropriately activated, all virtual methods need to be activated when an uninstrumented module is loaded. However, we believe this case should be rare in practice and thus do not activate all virtual methods in our current implementation.

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 (see details in §2.2.1). 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 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.

A better design would be activating exception handlers when their corresponding `try` block is executed for the first time. When multiple `try/catch` blocks exist in a single function, this scheme may activate less exception handler than the scheme we have implemented. We leave the implementation as future work.

3.2.3 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 and process forking.

Dynamic linking. 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 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 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 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.

Besides, πCFI supports dynamic library unloading and the detailed process is the same as what MCFI does (details in §2.3).

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 3.3). 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. Next, `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 3.3). 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.

3.3 Implementation

The πCFI toolchain is based on MCFI's toolchain. πCFI has around 300 lines of code more than MCFI's compiler to identify function address-taking instructions and insert calls to `patchstub_at` before these instructions (detailed in §3.2). In addition, about 320 lines of code were added to the MCFI runtime to implement the secure patching processes.

3.4 Evaluation

We used the MCFI's test environment configuration to evaluate πCFI, so please refer to §2.6 for details.

3.4.1 ECFG Statistics

We compiled and instrumented all 19 SPECCPU2006 C/C++ benchmark programs and 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

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 3.1: ECFG statistics of SPECCPU2006 C/C++ benchmarks.

the most address-activation operations (i.e., the worst case). The results are shown in Table 3.1. 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, 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 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 studying 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 3.11 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 did observe an outlier, `403.gcc`, when tested over the g23 data set, whose address activation curve is drawn in Figure 3.12. 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`.

To demonstrate that process forking would cause different CFG growths for the parent and child processes in a long running program, we used πCFI to protect an Nginx server and used the sever to host a WordPress site. Then, we used almost all features of WordPress for a session of about 20 minutes. Table 3.2 shows the address activation results. We configured Nginx to use two processes: the master process was responsible for initialization and handling administrators'

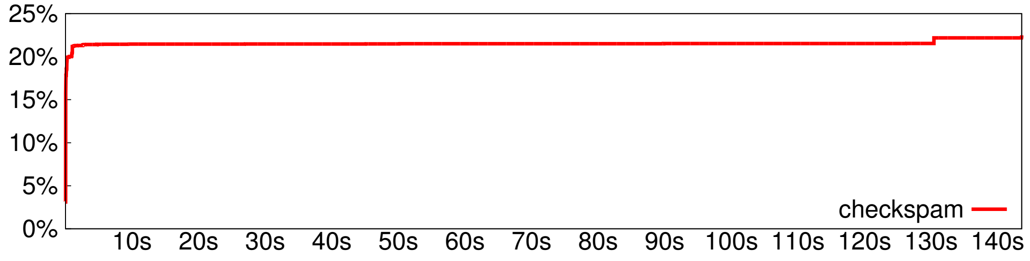


Figure 3.11: Growth of activated target addresses for 400.perlbench.

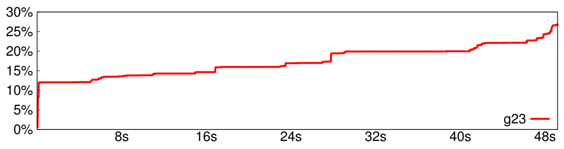


Figure 3.12: Growth of activated target addresses for 403.gcc.

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 3.13 shows the target activation growth curve for the worker process. Similar to other tested programs, the percentage quickly stabilized.

3.4.2 Performance Evaluation

Execution Time Overhead

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 experimental results of the performance overhead of πCFI, including runtime and space overhead. Of the two, having a small runtime overhead is much more important.

The runtime-overhead results of SPECCPU2006 are presented in Figure 3.14. 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. Compared to MCFI, πCFI causes a small increase of runtime overhead, due to address-activation operations and execution of nops after patching.

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

Table 3.2: ECFG statistics of the Nginx HTTP server's master and worker processes.

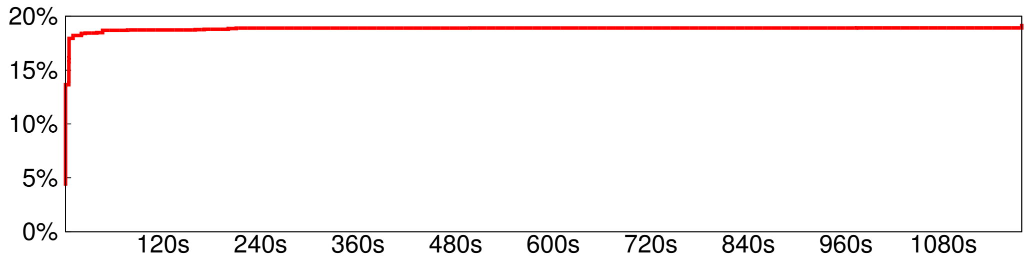


Figure 3.13: Growth of activated target addresses for Nginx.

We also tested the performance overhead of a πCFI-hardened Nginx-1.4.0 server, compiled at the O2 optimization level. The πCFI-protected Nginx run nearly as fast as the native version.

Space Overhead

πCFI may insert more code than MCFI, so its space overhead is slightly higher than MCFI, shown in Figure 3.15. On average, πCFI bloats the code by around 22.9%, 0.3% more than MCFI. However, the memory footprint increase is still negligible. In addition, the code size for Nginx increases by 22.8%, similar to SPECCPU2006 programs.

3.5 Future Work

Target deactivation. 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

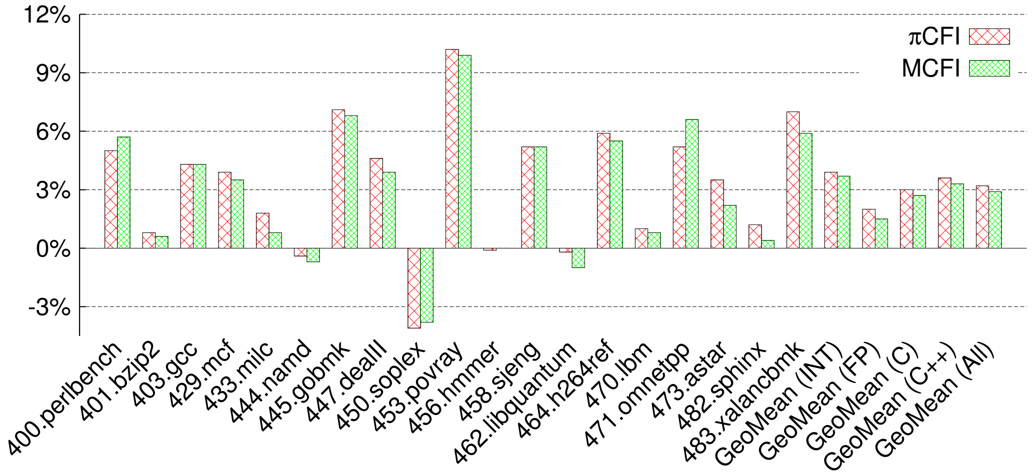


Figure 3.14: πCFI runtime overhead on SPECCPU2006 C/C++ benchmarks.

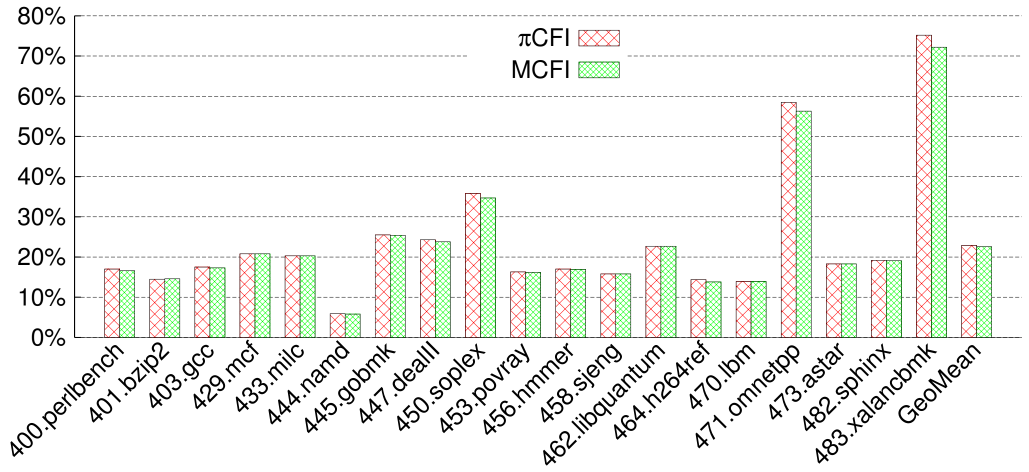


Figure 3.15: πCFI code size increase on SPECCPU2006 C/C++ benchmarks.

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.

Portability. We would like to investigate how πCFI can be implemented on other CPU 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 POWER.

Physical code memory saving. Current πCFI support for process forking directly copies the code pages in both the parent and child processes, thus doubling the physical code memory. Therefore, another potential direction of future work is how to improve πCFI to reduce or eliminate the extra physical code pages.

3.6 Summary

This chapter presents πCFI, a general method for lazily computing per-input CFGs based on a statically generated CFG generated by MCFI. Besides MCFI's instrumentation, πCFI inserts more code to explicitly take addresses of functions and register them to the ECFG dynamically. As the experiments show, πCFI can effectively reduce the available indirect branch edges for attackers.

The content of this chapter is based on our previously published paper [29] with no major changes.

Chapter 4

RockJIT

4.1 Overview

In this chapter, we discuss how to enforce πCFI for Just-In-Time (JIT) compilers. (Enforcing πCFI is stronger than enforcing MCFI, so we do not discuss MCFI enforcement separately.) JIT engines dynamically emit code to writable memory pages, and then execute the native code for speed. Traditionally, JIT engines suffer from both code injection and reuse attacks (e.g., [46, 47]). However, it is not straightforward to extend πCFI to JIT engines, because of the following challenges:

- Our goal is to enforce CFI for the JIT engine and the JITted code, so we need to compute a single CFG for both parts, neither of which is trusted. The (sub)CFG for the JIT engine can be generated using source-level information acquired during a trusted compilation process, but the JITted code cannot use this process as its compilation process is not trusted. Since the JIT engine itself is subject to malicious manipulation, how to securely and efficiently generate a (sub)CFG for the JITted code and merge it with the JIT compiler's CFG?

- Since JIT engines emit code on-the-fly, which might be corrupted by attackers, how to securely install, modify and delete the JITted code?

We propose a general approach called RockJIT to solve the above problems. Noticing that JIT compilers share a common architecture that naturally separates the CFG part of the JIT engine

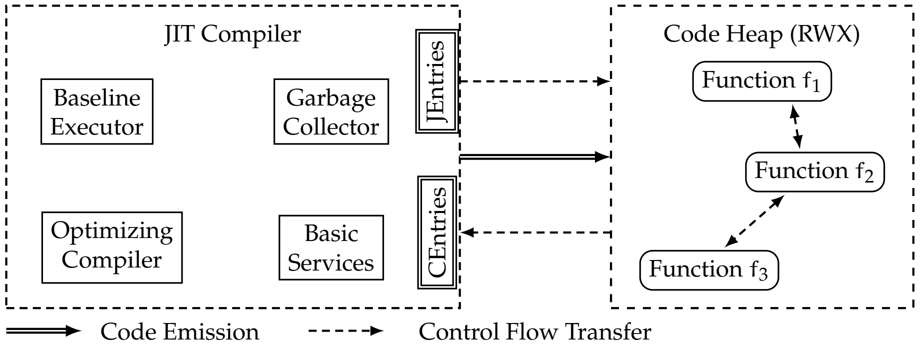


Figure 4.1: The common architecture of modern JIT compilers.

and that of the JITted code, we enforce πCFI on the JIT engine while coarse-grained CFI for the JITted code. We reuse the πCFI memory layout and export JITted code installation, deletion and modification trampolines to JIT engines to invoke. Any JITted code manipulation is delegated to the πCFI runtime and verified on-the-fly.

4.2 System Design

4.2.1 Common JIT Architecture

We investigated a range of JIT compilers, including Google V8 (JavaScript), Mozilla TraceMonkey (JavaScript), Oracle HotSpot (Java), Facebook HHVM (PHP), and LuaJIT (Lua). We found that their architectures share many commonalities and can all be represented by the diagram in Figure 4.1. A JIT compiler emits JITted code in the code heap and executes it. The code heap is readable (R), writable (W), and executable (X). A typical JIT compiler contains the following major components:

Baseline Executor. When a program starts running, its execution is the job of the baseline executor. Oftentimes, the baseline executor is an interpreter, which is easy to implement but slow. For instance, HotSpot has an interpreter that interprets Java bytecode. The baseline executor may have a different implementation from an interpreter. For example, the baseline executor of V8 compiles JavaScript source code directly to unoptimized native code.

Optimizing Compiler. During the execution of a program by the baseline executor, the JIT compiler performs runtime profiling to identify hot code and to identify types in the case of dynamically typed languages. Based on the runtime profile, the optimizing compiler generates optimized native code. JIT engines can have quite different designs for the optimizing compiler. For example, V8 profiles method execution and optimizes a whole method at a time. However, TraceMonkey profiles execution paths (e.g., a hot loop) and performs trace-based optimization [48] .

Garbage Collector. Managed languages provide automatic memory management, which is supported by a garbage collector. Most garbage collectors implement common algorithms such as concurrent mark and sweep.

Basic Services. The JIT compiler also provides runtime services, including support for debugging, access to internal states for performance tuning, foreign function interfaces for enabling interoperability between managed languages and native code.

For performance, all JIT compilers we inspected are developed in C/C++. Since the calling convention of C/C++ is different from that of JITted code, which is JIT-compiler specific, JIT compilers introduce interfaces to allow context switches between the code of the compiler and JITted code. In Figure 4.1, the interfaces are depicted as `JEntries` and `CEntries`; both are essentially indirect branches. `JEntries` transfer control to JITted code and `CEntries` transfer control to the JIT compiler. As an example of `JEntries` in V8, the initial control transfer from the JIT compiler to the code heap is through an indirect call (`JEntry`) in a code stub called `JSEntryStub`. As an example of `CEntries`, V8 provides services (or functions) such as JavaScript object creation and object property access. When JITted code invokes these services, the control can be first transferred to a stub called `CEntryStub` with a register containing the address of the target service function. Within `CEntryStub`, an indirect call (`CEntry`) through the register is executed to transfer the control to the service function. Moreover, it should be noted that both `CEntries` and `JEntries` could be dynamically generated (e.g., when emitting JITted functions that directly invoke `CEntries` to efficiently call a JIT-engine service).

4.2.2 RockJIT Architecture

RockJIT transforms the common JIT architecture to the one visualized in Figure 4.2, based on the memory layout in Figure 3.3. The πCFI runtime provides services to a JIT compiler and monitors its security. An existing JIT compiler such as V8 should be modified to cooperate with πCFI's runtime. It is then compiled and instrumented by πCFI's compilation toolchain. The compiled JIT engine is loaded by πCFI into the SFI sandbox. After loading, the πCFI runtime generates an SCFG for the JIT compiler based on the meta-information in the module, constructs Bary and Tary tables that encode the SCFG and ECFG, and starts execution of the JIT compiler. Same as how πCFI handles secure code patching, all executable memory pages in the sandbox are made non-writable, and they are mapped to outside-sandbox writable pages. As shown in the figure, the code heap's physical pages are also mapped from the shadow code heap outside of the sandbox. Therefore, JITted code installation, modification and deletion are all conducted by the πCFI runtime on the shadow code heap.

Since it is possible that the attackers could modify the JITted code before it is installed, RockJIT performs online *verification* on the native code to check a set of properties (detailed in §4.3.1) for security. If the verification succeeds, RockJIT installs the new code in the shadow code heap and updates MCFI tables by taking the new code into account.

4.2.3 RockJIT CFG Generation

RockJIT enforces control-flow integrity on both the JIT compiler and JITted code, but applies different precision on those two parts. For the JIT compiler, RockJIT uses the πCFI toolchain to enforce a per-input fine-grained CFG. In contrast, the CFG for JITted code is coarse-grained in the sense that all its indirect branches share a common set of targets and no address activation is conducted on-the-fly. The JIT compiler is modified to emit not only instrumented JITted code, but also information about indirect-branch targets. The verifier then deduces the coarse-grained CFG for the new code and combines it with the old CFG.

The approach of hybrid CFI precision in RockJIT is the result of a careful consideration of both security and performance. First, the JIT compiler's code is mostly where the majority of

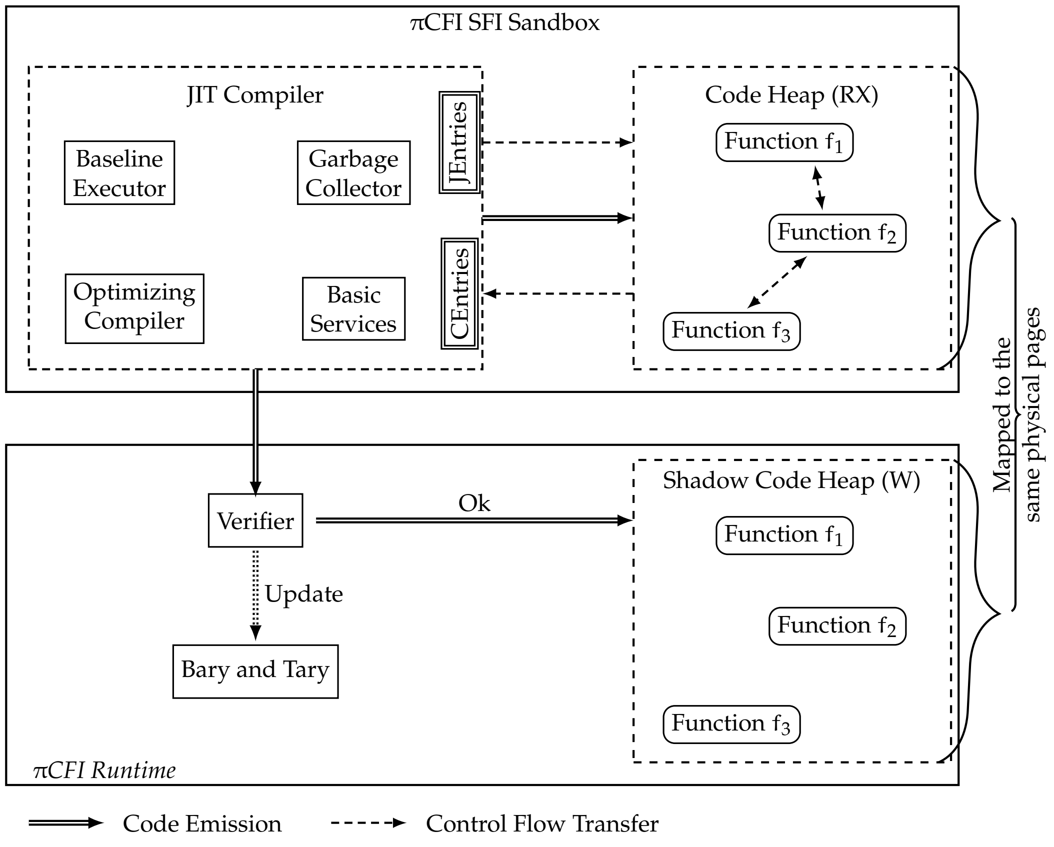


Figure 4.2: The architecture of RockJIT.

the code is and contains dangerous system call invocations. Since its code is statically available, constructing a fine-grained CFG offline and a per-input CFG online for the JIT compiler increases security substantially as recent work has shown that coarse-grained CFI can still be easily attacked by ROP attacks [14–16]. On the other hand, JITted code is frequently generated on-the-fly and for performance it is important that verification and new CFG generation do not have high overhead. Verification and CFG-generation algorithms for coarse-grained CFI are simpler and thus can run much faster. However, a big concern of coarse-grained CFI for JITted code is that it might jeopardize security. We do not believe that is the case because JITted code should not contain dangerous instructions such as system calls, a property that is enforced by RockJIT's verifier; such instructions are required in an attack. JITted code can still request system-call services from the JIT compiler, but the JIT compiler is hardened through πCFI: security is preserved as long as sufficient checks are placed along per-input CFG paths. Moreover, enforcing coarse-grained CFI on the JITted code makes the RockJIT approach general to handle all kinds of JIT-compiled languages (e.g., Java bytecode, PHP, etc.) for which fine-grained CFG generation may be hard.

One point worth mentioning is that, thanks to the verifier (in §4.3.1), the JIT compiler is not in the TCB even though it performs runtime code generation. The native code generated by the JIT compiler is first checked to obey a set of safety properties before installed. The verifier is in the TCB but it is much smaller than the JIT compiler.

4.3 JITted Code Manipulation

The code heap maintained by a JIT compiler is where code is dynamically managed. It consists of multiple code regions such as functions. A JIT compiler dynamically installs, deletes, and modifies code regions. New code regions are frequently generated by the compiler and installed in the code heap. When a code region is no longer needed, the JIT compiler can delete it from the code heap and reuse its memory for future code installation. Runtime code modification is mostly used in performance-critical optimizations. As an example, *inline caching* [49, 50] is a technique that is used in JIT compilers to speed up access to object properties. In this technique, a JIT compiler modifies native code to embed an object property such as a member offset after the property has been accessed for the first time, avoiding expensive object-property access operations in the future. Another example of runtime code modification happens in V8 during code optimization. V8 profiles function and loop execution to identify hot functions and loops. It performs optimization on the hot code to generate an optimized version. Afterwards, runtime code patching is performed on the unoptimized code to transfer its control to the optimized version through a process called on-stack replacement [51].

Since RockJIT enforces CFI, it is necessary to check security for each step of runtime code installation, deletion, and modification. In §4.3.1, we present how verification is performed when a new piece of code is installed. The process for code deletion and modification has only small differences; we leave their discussion to §4.3.2 when we discuss the detailed steps for runtime code manipulation. In all cases, we take the Google V8 JavaScript engine as an example, since we have modified a version to implement RockJIT.

4.3.1 JITted Code Verification

The verifier in general maintains three sets of addresses that are code addresses in the code heap:

- **Pseudo-instruction start addresses (**PSA**).** This address set remembers the start addresses of all *pseudo-instructions*. We define a pseudo-instruction as: (1) a *checked indirect branch*, which is a CFI check instruction sequence (detailed in §4.4) for checking a register `r` immediately followed by an indirect branch through `r`; or (2) a *sandboxed memory write*, which is a `0x67`-prefixed memory write for SFI (see §2.3.2); or (3) an instruction that is neither an indirect branch nor an indirect memory write.

- **Indirect branch targets (**IBT**).** This address set records all possible indirect branch targets.

- **Direct branch targets (**DBT**).** This address set remembers all direct branch targets.

The critical invariant of the three sets is $\mathrm{IBT} \cup \mathrm{DBT} \subseteq \mathrm{PSA}$. That is, all indirect and direct branch targets must be start addresses of pseudo-instructions. With this invariant, it is impossible to jump to the middle of an instruction where system call instructions may hide. Furthermore, it is impossible to transfer the control to an indirect branch or a memory write without executing its preceded MCFI check, which is necessary for SFI.

The three sets can be built incrementally with the installation of new code. Initially, they are all empty sets when the code heap contains no code. When a new code region is installed, the verifier updates the three sets by computing PSA', IBT' and DBT' after taking new code into consideration. For instance, direct branch targets (DBT') can be computed from the code alone. PSA' can be computed by disassembling the JITted code, while IBT' can be computed by modifying the JIT compiler to emit legal indirect branch targets as metadata.

With the new address sets, the verifier checks $\mathrm{IBT}' \cup \mathrm{DBT}' \subseteq \mathrm{PSA}'$ and the following constraints on the new code:

C1 Indirect branches and memory-write instructions are appropriately instrumented. In particular, only checked indirect branches and masked memory writes are allowed.

C2 Direct branches jump to addresses in DBT$'$. This ensures that the new code respects DBT$'$.

C3 The code contains only instructions that are used for a particular JIT compiler. This set of instructions is usually a small subset of the native instruction set and can be easily derived by inspecting the code-emission logic of a JIT compiler. Importantly, this subset cannot contain system calls and privileged instructions.

Next, we present some implementation details about a verifier we constructed for V8. Our coarse-grained CFI policy allows each indirect branch in the JITted code to target any pseudo-instruction for simplicity, so IBT $=$ PSA. The address sets are implemented by bitmaps for fast look-ups and updates. Each bitmap maps a code address to one if and only if that address belongs to the corresponding set, otherwise zero.

In addition, the speed of verification is of practical importance. Since V8 performs frequent code installation, a slow verifier can negatively impact the performance non-trivially. For example, NaCl-JIT [43] includes a disassembly-based verifier and it reports 5% overhead for the verification alone. We adopt an approach based on Deterministic Finite Automata (DFA) following RockSalt [52]. It performs address-set updates and constraint checking in one phase without doing full code disassembly. Our verifier incurs only 1.3% overhead for the verification.

In detail, we wrote a 2776-line Python script (`instr.py`) to enumerate all possible allowed instruction encoding as byte sequences. Next, we used another 219-line Python script (`trie.py`) to build a trie structure [53], which was next converted to a DFA by another 185-line Python script (`trie_to_c.py`). Both `trie.py` and `trie_to_c.py` scripts were modified from Seaborn's code [54]. The DFA has 411 states in total, including 17 acceptance states (e.g., accepting a direct call) and 1 rejection state. The verifier iterates all JITted code using the DFA. When a direct branch is matched, it records its jump target; when a checked indirect branch, a masked memory write, or one allowed instruction is matched, it moves forward. In the above cases, the pseudo-instruction boundaries are recorded as well. The verification fails whenever the DFA reaches the rejection state (e.g., due to an illegal instruction). After all code bytes have been matched, the verifier updates the address sets and checks that DBT$' \subseteq$ PSA$'$. When the verification succeeds, constraints C1–C3 are respected by the JITted code.

Recall that our threat model (§1.3.1) does allow attackers to write arbitrary memory pages in the sandbox that are writable, so after the code is emitted in the sandbox and before it is copied outside of the sandbox for verification, the attackers might corrupt it. However, the corrupted code still needs to pass the verification. If it passes the verification, the CFI property cannot be violated.

4.3.2 JITted Code Installation, Deletion, and Modification

In RockJIT, a JIT compiler cannot directly manipulate the code heap, which does not have the writable permission. Instead, RockJIT provides services to the JIT compiler for code installation, deletion, and modification. One worry for runtime code manipulation is thread safety: one thread is manipulating code, while another thread may see partially manipulated code. This is more general than πCFI, which carefully arranges instructions so that code modification always changes only a single instruction at a time. We next discuss the detailed steps involved in RockJIT's code manipulation and how thread safety is achieved.

Code installation. For code installation, the JIT compiler invokes RockJIT's code installation service and sends a piece of native code, the target address where the native code should be installed, and meta-information about the code for constructing new address sets. The code-installation service then performs the following steps:

1. The verifier performs verification on the code and updates the address sets to PSA′, IBT′, and DBT′.

2. If the verification succeeds, the code is copied to the shadow code heap at an address computed from the start address where the code should be installed.

3. The runtime tables used by MCFI are updated to take into account the new code. Since coarse-grained CFI is enforced on JITted code, only information in IBT′ is needed to update the tables.

There are a couple of notes worth mentioning about the above steps. First, the verification of benign programs is expected to succeed if there are no bugs in the JIT compiler. A verification failure indicates a bug that should be fixed. Second, it is important that the MCFI tables are updated after copying the code, not before. During the copying process, the code becomes partially visible to the JIT compiler as the code heap is mapped to the same physical pages as the shadow code heap. However, since the MCFI tables have not been updated yet, no branches can jump to the new code, avoiding the situation in which one thread is installing some new code and another thread branches to partially installed code.

Code deletion. Deletion of JITted code is similar to library unloading, with the only difference being that before deactivating all targets in the code being deleted, RockJIT should make sure there is no direct branch instruction targeting the code that is to be deleted.

Code modification. If the new code region has the same internal pseudo-instruction boundaries and native instruction boundaries as the old code region, only the native instructions are modified, and the new code passes verification, RockJIT follows NaCl-JIT's approach to replace the old code with the new code. Otherwise, code modification is implemented as a code deletion followed by a code installation.

4.4 Modification to a JIT compiler

Existing JIT compilers need to be modified to work with RockJIT. We next report our experience of adapting Google's V8 JavaScript engine (3.29.88.19). To adapt V8's x64 source, we modified 1934 lines of its source code: 1914 lines were changed to make it generate πCFI-compatible code and invoke RockJIT's services for runtime code manipulation; 20 lines were added for bad type casts (details in §2.2.2) that prevent sound SCFG generation. This experience demonstrates that modifying an existing JIT compiler to work with RockJIT requires modest effort. Most of the changes to V8 were in its code-emission logic to make the generated code compatible with πCFI:

- Code-emission functions that generate indirect branches were modified to generate checked indirect branches. We could directly use MCFI's instrumentation (i.e., the transactions) to rewrite the JITted code, but for coarse-grained CFG enforcement, we use a simplified CFI check implementation. First, we reserve byte `0xf4` so that it never appears in any Bary or Tary IDs. Then, for each indirect branch target in JITted code, we map it to `0xf4` in the Tary

```
1       cmpb $0xf4, %gs:(%r10)
2       jne  -3
3       jmpq *%r10
```

Figure 4.3: Indirect branch instrumentation in JITted code.

table; other code addresses are therefore mapped to byte values that are never `0xf4`. This is similar to how MCFI supports interoperability (§2.4).

Next, we instrument each JITted indirect branch in a way similar to Figure 4.3, assuming register `r10` contains the target. The comparison at line 1 simply checks whether the target's corresponding Tary byte is `0xf4`. If so, the next `jne` instruction is a nop, which leads to the actual indirect control transfer at line 3. Otherwise, the `jne` instruction jumps back three bytes, landing in the middle of the comparison instruction's last byte, `0xf4`. `0xf4` happens to be the encoding for the `hlt` instruction, resulting in termination of the JIT engine.

- Code-emission functions for indirect memory writes were modified to generate masked memory writes. The sandbox resides in the [0, 4GB) memory. Therefore, an indirect memory write should be prefixed with `0x67` to override the 64-bit address to 32-bit (details in §2.3.2).

Since V8 also emits `JEntries` and `CEntries` on-the-fly, RockJIT provides services for V8 to securely install those `JEntries` and `CEntries` as well as their type signatures to enable SCFG generation.

Another part we modified was to accommodate online code patching. When V8 emits certain optimized native code, it reserves some bytes in the code in anticipation of future code patching (for a process called deoptimization). The original V8 reserves 13 bytes for such purpose. RockJIT needs more bytes because of extra MCFI checks; we had to reserve 24 bytes instead.

Finally, changes were made to V8 to invoke code installation, deletion, and modification services provided by RockJIT at appropriate places.

Compared to related work, RockJIT changes around 60% less code than NaCl-JIT, which changed over 5,000 lines of code for the x64 version of V8. NaCl-JIT requires more changes because: (1) it disallows the mix of code and data in the JIT-compiled code and V8 has to be

	IBs (with matching targets)	IBTs	EQCs	Avg IBTs / IB	Avg IBs / IBT
V8	35775 (29609)	116,919	9,696	808	205

Table 4.1: Equivalence classes for Google V8 JavaScript compiler.

changed to separate code and data; RockJIT's CFI allows the mixture of code and data as long as data cannot be reached from code with legal control flow; (2) NaCl-JIT uses the ILP32 programming model on x64, while the native V8 uses LP64 model; therefore, it has to change nearly the entire code-emission logic.

4.5 Evaluation

We compiled the modified Google V8 JavaScript compiler to a standalone executable using the πCFI toolchain and measured the CFG statistics and performance overhead with the same system configuration as reported in §2.6.

4.5.1 SCFG and ECFG Statistics

RockJIT supports fine-grained SCFGs for the JIT compiler. Table 4.1 presents the details of the SCFG extracted for V8. Similar to SPECCPU2006 C++ benchmarks, thousands of equivalence classes are supported, and the average number of targets of indirect branches and average number of indirect branches targeting an address are much less than coarse-grained CFI, which could be the number of indirect branch targets and the number of indirect branches, respectively.

We ran V8 on three benchmark suites: Sunspider 1.0.2, Kraken 1.1, and Octane 2, and collected ECFG statistics (as percentage compared to the SCFG) for those benchmark suites listed in Table 4.2. The meanings of column names are the same as those of Table 3.1. 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

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%

* Please refer to table 3.1 for meanings of the column names.

Table 4.2: ECFG statistics of the Google V8 JavaScript engine.

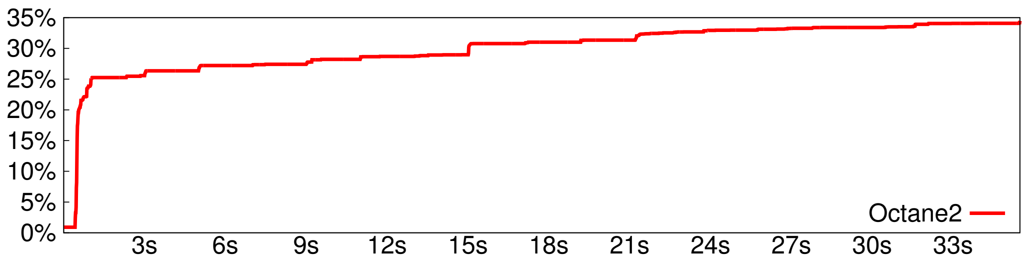


Figure 4.4: V8 CFG growth for Octane 2.

significantly more addresses than those benchmarks. The ECFG growth curve of V8 when tested over Octane 2 is shown in Figure 4.4, 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 (e.g., Figure 3.11 and 3.12).

Compared to NaCl-JIT, which enforces a form of coarse-grained CFI on V8's code, RockJIT's SCFG removes about 99.7% indirect branch edges from NaCl-JIT's CFG, and the ECFG generated on Octane 2 eliminates over 99.9% indirect branch edges.

4.5.2 Performance Overhead

Execution Time Overhead

As already discussed, πCFI slows down program execution. We measured the slowdown of πCFI-instrumented V8 over Octane 2 benchmarks, which is shown in Figure 4.5. On average, 12.1% runtime overhead is incurred by πCFI, while 11.7% by MCFI. As analyzed before, πCFI costs a bit more time than MCFI due to online address activation and patched nops.

In general, RockJIT's performance overhead are due to five major contributors: separation of

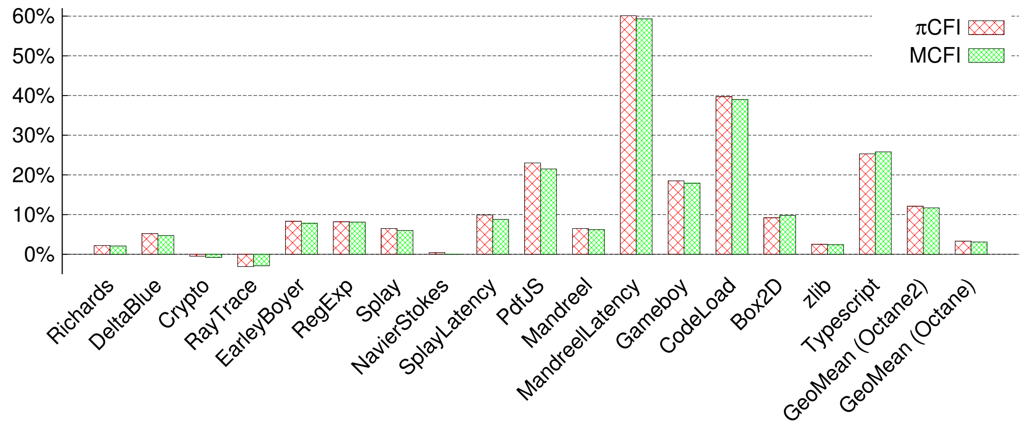


Figure 4.5: πCFI and MCFI overhead on Octane 2 with Google V8.

Separation of the code heap and shadow code heap	6.2%
MCFI instrumentation of V8's code	2.2%
πCFI online activation	0.4%
CFI instrumentation of JITted code	2.0%
Online verification	1.3%
Total	12.1%

Table 4.3: Performance overhead contributors to RockJIT-hardened V8.

the code heap and shadow code heap, MCFI instrumentation of the JIT compiler's code, πCFI online address activation, instrumentation of the JITted code, and verification. Table 4.3 shows the performance overhead of each contributor over Octane 2 benchmarks. These overhead results were generated by disabling overhead contributors one at a time.

Also, we separately calculated runtime-overhead results for the subset of benchmarks that were included in Octane 1 (the predecessor of Octane 2) since related works use Octane 1 for evaluation. πCFI incurs only 3.1% overhead over them on average. Compared to other JIT-compiler hardening works, such as NaCl-JIT [43], librando [55], and SDCG [46], πCFI incurs less overhead and provides better security.

Space Overhead

In terms of code bloat, the code size of V8 increases by around 39.4% after the πCFI instrumentation, in which 38.2% is due to MCFI checks and 1.2% is for πCFI's address-taken instrumentation.

The size of JITted code generated for the Octane 2 benchmarks is 18.2% larger with the CFI instrumentation presented in §4.4.

4.6 Future Work

RockJIT for Trace-JIT. V8 adopts the traditional method-JIT technology to emit JITted code one function at a time, while other JIT engines such as FireFox SpiderMonkey may adopt trace-based JIT [48] to generate code traces, which might consist of code parts from multiple functions. Although we consider there would not be any technical challenges when extending RockJIT to trace-based JIT engines, we still believe it would be interesting to confirm this and secure both kinds of JIT engines.

Parallelizing the πCFI runtime. Note that the separation of code heap and shadow code heap incurs the single largest performance overhead shown in Table 4.3. We suspect that the current implementation of the πCFI runtime may be the culprit, because it uses a single global lock to synchronize all threads. Given that V8 is a multi-threaded JIT engine that has separate threads for concurrent compilation and garbage collection, both of which need to enter the runtime for code installation and memory reclamation, respectively, the lock may be highly contended. We plan to investigate the issue to further reduce the performance overhead.

CFG precision improvement for the JITted code. RockJIT enforces coarse-grained CFI for the JITted code to balance performance and security. In the future, it might be worth exploring new methods for generating fine-grained CFGs for the JITted code without jeopardizing the performance. After all, the more precise the CFG is, the more security we gain from CFI.

Full browser CFI enforcement. In practice, JavaScript engines are often part of web browsers, which also include many other libraries for page rendering, multimedia, etc. Those libraries may also be JIT engines, such as the Adobe Flash Player in Chrome. Conceptually, it should be straightforward to extend RockJIT and πCFI to full browsers, and we leave this to future work.

4.7 Summary

This chapter presents a general approach to enforcing fine-grained CFI (πCFI and MCFI) for Just-In-Time (JIT) engines by (1) using parallel memory mapping to monitor and securely modify JITted code pages and (2) separately generating the CFG parts for the JIT engine and JITted code and then merging them into a single CFG.

The content of this chapter is based on our previously published work [28, 29]. Note that the JITted code instrumentation in [28] is the same as that in [27], while this chapter and [29] share a simpler version of the JITted code instrumentation described in §4.4. In addition, the V8 version used in [28] is older than the version mentioned in this chapter and [29].

Chapter 5

Security Analysis

We discuss security benefits of previously proposed CFI schemes and focus on πCFI, since it provides better protection than MCFI thanks to finer-grained CFGs while incurring comparable overhead. In general, 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 system 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 as discussed in §4. Attackers may still inject code into the JITted code, but the injected code never violates CFI because of online code verification. For example, the injected code can never contain any system call instruction.

For code-reuse attacks (e.g., ROP), πCFI mitigates them by enforcing a fine-grained per-input CFG, which provides multiple security benefits. First, the number of ROP gadgets that current tools can recognize is decreased. For instance, a ROP gadget finding tool `rp++` [56] can only find around 4% of the ROP gadgets in instrumented SPECCPU2006 benchmarks compared to the results in native binaries, and the reason is CFI disables those ROP gadgets that start from the middle of instructions. 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 (e.g., `execve`). Third, when there is unreachable code with respect to a concrete input, the per-input ECFG is likely to exclude the unreachable code. For instance, if a libc function is never called in an application's source code (including libraries), 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 calling 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.

5.1 Mitigation of Advanced Attack Forms

In this section, we describe recently proposed advanced attack forms of code injection and code reuse, and discuss how πCFI could mitigate these attacks.

5.1.1 Just-In-Time Code Reuse

Just-In-Time Code Reuse [47], or JIT-ROP, is an advanced ROP attack specially targeting JIT engines. JIT-ROP firstly exploits a memory leak bug of a JIT engine and finds a code byte address. Then, it scans the memory page of that leaked code byte to see if there are pointers inside the page that references other code pages. If so, it follows the pointers and recursively explores other executable memory pages. After harvesting sufficient memory pages, JIT-ROP scans those pages again for ROP gadgets, and uses a specially built compiler that considers the ROP gadgets as instructions to compile the attack payload. Finally, it executes the attack payload by jumping to the first gadget.

πCFI can mitigate JIT-ROP by making it harder to find usable gadgets and chain them. With πCFI, gadgets cannot begin from the middle of instructions; instead, they have to start from a valid indirect branch target; a function address, for example. This implies that gadgets in πCFI are much longer and involve more side effects, which make exploit writing more difficult. Moreover, the fine-grained per-input CFG makes attacks harder by restricting which gadgets can be connected to which other gadgets. As our experiments have shown in §4.5.1, πCFI significantly reduces the number of indirect branch edges, which are essential to chain the gadgets.

5.1.2 JIT Spraying

JIT compilers have large attack surfaces, since the input program can be fully controlled by an attacker. Specifically, a JIT spraying attack [57] takes advantage of the often predictable code-emission logic in the JIT compiler. The attacker crafts an input program with special embedded constants and uses a vulnerability in the JIT compiler to hijack the control flow to execute those constants as malicious code. To illustrate this point using JavaScript, suppose the input code is "`x = x ∧ 0x3C909090`", where ∧ is JavaScript's `xor` operator. A JavaScript compiler may generate native code for implementing the `xor` operation, in which we assume the constant `0x3C909090` is encoded literally. Therefore, the byte sequence on x64 for encoding an `xor` operation is as follows, assuming `%eax` holds the value for `x`.

```
35 90 90 90 3c:   xorl 0x3C909090, %eax
```

Now suppose the JavaScript compiler has a vulnerability that enables the attacker to control the program counter. Because x64 has a variable-length instruction set, the attacker can change the control flow to point to the middle of the above instruction and start the execution of a totally different instruction stream from the intended. For example, if the program counter is changed to point to the first `0x90` in the above, the next instruction to execute is a nop (the encoding of `0x90`), followed by other instructions not intended in the original program. Note that the constant `0x3C909090` above is under the control of the attacker, who can put any constant there for executing arbitrary code.

Modern operating systems deploy ASLR, which makes it hard for the attacker to locate the absolute addresses of constants in instructions such as xor. The attacker, however, can spray many copies of the same code in memory to increase the chance of a successful attack on the JIT; this is why it is called JIT spraying. Real JIT spraying attacks have been demonstrated, for example, on the JavaScript engine of the Safari browser [58] and Adobe's Flash Player [59].

One observation about JIT spraying attacks is they involve both the JITted code and the JIT compiler. The attacker takes advantage of the fact that the JITted code is often predictable for a given piece of source code. Furthermore, there must be a vulnerability in the JIT compiler so that the control can be transferred to the middle of an instruction to start an unexpected and harmful code sequence. Given this observation, one natural defense is to randomize code generation to make the generated native code less predictable. This approach has been explored by systems such as librando [55] and others. The downside is that it provides only a probabilistic defense. Instead, RockJIT (or πCFI) hardens both the JIT compiler and the JITted code using control-flow integrity so that it is impossible to transfer the control to the middle of instructions. As a result, unexpected instructions can never be executed, making JIT spraying impossible in RockJIT-protected JIT compilers.

5.1.3 Counterfeit Object-Oriented Programming

Counterfeit Object-Oriented Programming (COOP [60]) 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 (§2.2.1) and performs online activation of virtual methods, COOP is made harder on πCFI-protected C++ applications.

5.1.4 Control-Flow Bending

Control-Flow Bending (CFB, [61]) is a recently proposed general methodology for attacking conventional CFI systems that statically generate CFGs, including MCFI. 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 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 be prevented as well, 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 functions' callers do not have as many call sites as those functions, what the attackers can do becomes more restricted. The same technique could be implemented in the πCFI toolchain for all leaf functions, and potential dispatchers might be appropriately inlined to be such leaf functions whose execution therefore never changes its own return address.

5.1.5 Control Jujutsu

Control Jujutsu [62] is another recently proposed attack form targeting fine-grained CFI implementations including MCFI. At a high level, the attacker firstly finds special indirect call instructions, called Argument Corruptible Indirect Call Sites (ACICS), whose targets and arguments can both be controlled by attackers. Then, by using those ACICS gadgets that are close to critical system calls (e.g., `execve`) and carefully crafting those arguments, the attackers may achieve arbitrary code execution.

Fortunately, πCFI can mitigate such attacks by reducing the number of targets of ACICS gadgets. For example, the authors demonstrate a proof-of-concept attack against Nginx by redirecting an ACICS gadget to target `ngx_execute_proc` that later invokes `execve`. However, in πCFI-protected Nginx, since the CFGs of the master process and the worker process are different, `ngx_execute_proc` is never activated by any normal inputs. Therefore, attackers need to first steer the execution to activate `ngx_execute_proc`, if possible, to attack πCFI-protected Nginx.

5.1.6 Sigreturn-Oriented Programming

Sigreturn-Oriented Programming [35], or SROP, 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 (πCFI stores the stack inside the sandbox as described in §2.5) 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 stored inside the sandbox, 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 a long period of time.

5.2 Comparison with Deployed Defenses

5.2.1 Stack Cookie

Stack Cookie [4] is widely deployed on all major OSes including Windows, Linux and OSX. Its basic idea is to store some random value before the return address on the stack when executing the function prologue and check the cookie's integrity before the function returns. If there is any sequential buffer overflow vulnerability in the program, the cookie will be changed before the return address is changed, and therefore the check before the return will raise an exception.

Compared to πCFI, Stack Cookie only protects function returns. However, if a bug allows the attackers to arbitrarily control stack addresses (e.g., the attacker can control the index to an array), Stack Cookie can be easily bypassed. In addition, Stack Cookie does not protect any function pointers in the heap such as virtual table pointers, therefore it provides no protection against

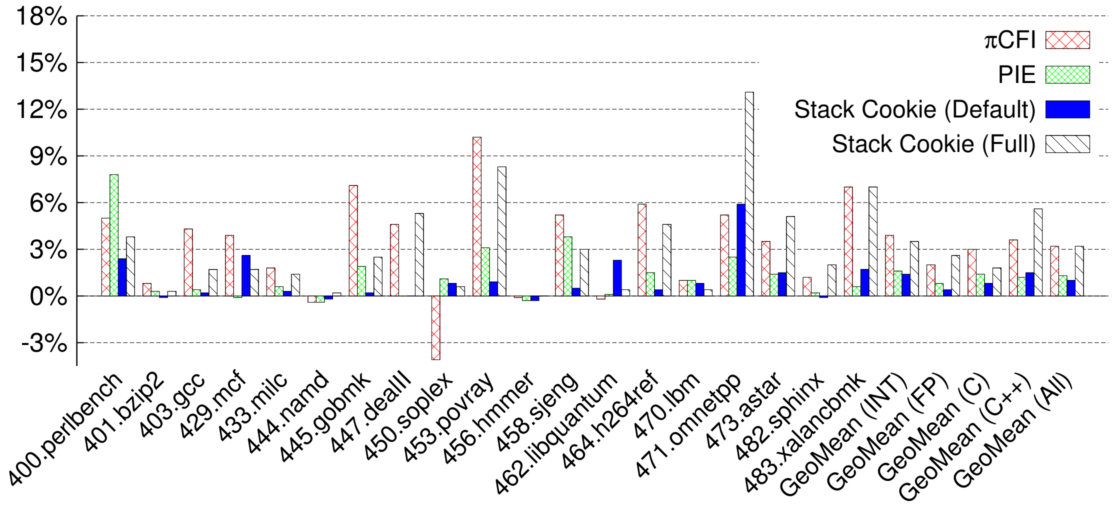


Figure 5.1: Runtime overhead comparison among πCFI, Position-Independent Executable (PIE) and stack cookie.

use-after-free exploits. On the other hand, πCFI protects the control flow in many more cases.

We benchmarked the performance overhead of Stack Cookie and compare it with πCFI. The detailed numbers are shown in Figure 5.1. "Stack Cookie (Default)" means that we used no extra compilation flag to compile the SPECCPU2006 benchmarks except "`-O3`". The compiler analyzes each function and if it is safe (e.g., a function does not manipulate any stack arrays), Stack Cookie will not be enabled for that particular function. For "Stack Cookie (Full)", we passed "`-fstack-protector-all`" to instrument each function with Stack Cookie's guard code.

As can be seen from the figure, πCFI incurs similar overhead to the full instrumentation of Stack Cookie, but more overhead than the default instrumentation. It might be possible to also perform Stack Cookie's analysis to disable πCFI instrumentation for certain functions and reduce πCFI's overhead, but concurrent return address corruption is then possible.

5.2.2 ASLR

Same as Stack Cookie, ASLR is deployed on the mainstream OSes by default. Different from Stack Cookie and πCFI, ASLR makes ROP harder by loading code modules at random memory addresses. However, as shown in §1.2, ASLR is vulnerable to memory leak, spraying and even brute-force cracking. In terms of performance overhead, ASLR on average incurs similar overhead to the default Stack Cookie protection, shown in Figure 5.1 as the "PIE" bars. In Linux, ASLR is disabled for the main executable but enabled for libraries. Therefore, we turned on ASLR for the main executable by passing "`-fPIE`" during compilation and compared the execution time with the default counterpart to get the above performance results.

5.3 Limitations of CFI and Future Research

As already discussed, CFI in general mitigates control-flow hijacking attacks, but data-only attacks [63] that always follow the CFG are out of CFI's protection scope, even if an ideal per-input CFG is enforced. For instance, the notorious "HeartBleed" [64] vulnerability of OpenSSL cannot be mitigated by any CFI. In detail, OpenSSL misses a check for a packet length so that a malicious client can read more data than it requires by sending a packet that claims its length is larger than the packet's actual length, resulting in possible private key leakage. Recently, Hu *et al.* [65] even demonstrated that data-only exploits can be automatically generated.

CFI can neither prevent logic errors. CFI assumes that the program as well as the program's CFG is "legal", but in practice, it is not always true. For example, Apple's "goto fail" bug [66] legitimately jumps over checks so that the verification would never fail. Even if CFI is enforced, that problematic control flow is still valid according to any CFG. Similarly, attacks such as SQL injection and cross-site scripting (XSS) are also logic errors in essence.

In summary, CFI is unfortunately not a silver bullet. In the future, more research is needed in the following areas to further improve software security.

- **Data-Flow Integrity** (DFI [67]) is effective at mitigating data-only attacks in general. DFI computes a legal static data flow graph for a program and inserts instrumentation to enforce that any runtime data flow should be specified in the data flow graph. DFI provides stronger protection than CFI, since CFI can be considered as enforcing DFI on only the control data. However, DFI still lacks many features that would make it practical: modularity, efficiency and interoperability. To make DFI practical, future research needs to first solve these problems, and we believe our work has laid the foundation for potential solutions.

- Enforcing **spatial and temporal memory safety** is a fundamental way of defeating memory corruption. Spatial memory safety (or bounds checking) requires that any pointer dereference should be within the bound that is established at memory object creation time. Example systems include SoftBound [68] and baggy bounds checking [69]. Temporal memory safety addresses object use-after-free issues by forbidding pointers to freed objects to be dereferenced. For example, CETS [70] associates each pointer and memory object a version number and compares the versions to determine whether a freed object is being dereferenced. The major problem of enforcing memory safety is the high performance overhead, therefore currently memory safety enforcement is more suitable to facilitate bug hunting during software testing (e.g., AddressSanitizer [71]). To be practical, current memory safety techniques need substantial overhauls for low overhead.

- **Information-hiding-based memory randomization** complements CFI by making it harder to precisely locate ROP gadgets, and recent research has gone far beyond ASLR by leveraging eXecute-no-Read (XnR) memory pages [72]. For example, Readactor [73] runs applications in virtual machines whose executable memory pages are set non-readable using hardware virtualization support, so the attackers cannot directly read the code to harvest gadgets. Then, it uses position-independent trampolines to relay indirect calls and returns to their targets and randomly place those trampolines to probabilistically eliminate code pointers in readable memory. Other recent memory randomization work includes [74–77]. However, these proposed techniques lack either efficiency or interoperability, therefore future research should make these techniques more practical.

- We believe **formal verification** is the only approach to secure software. Basically, the behaviors of software should be formally specified first, and a mathematical proof (typically machine-checkable) should be conducted on the software code to prove that the code correctly implements the specification. For example, seL4 [78] is a micro-kernel that has been formally verified to respect a specification. This area is under heavy research and we believe more research is needed to further lower the cost and improve the efficiency.

- **Quantitative security measurement** for CFI defenses is perhaps another important future research direction. Currently, there are three major quantitative security measurement metrics and all of them need to be improved to better reflect security enhancement. First, *ROP gadget reduction* is calculated to see how many ROP gadgets can be eliminated if a security defense approach is deployed. The baseline is calculated using existing ROP gadget finding tools, which only find simple gadgets (e.g., several instructions followed by an indirect branch). However, the baseline is not strong enough since new attack research keeps finding more sophisticated ROP gadgets (e.g., full functions in COOP [60]) that bypass the defense. As a result, it is reasonable to study how to semantically define ROP gadgets and find all equivalent ones (if possible) in a program as a strong baseline. Fortunately, Q [79] has provided inspiration in this direction. Second, the *AIR* [7] value is often calculated to measure how many targets can be reduced for each indirect branch on average. It is always less than one, and the more it approaches one, the better. MCFI reports a 99.97%+ AIR value, but it has been shown to be vulnerable to CFB [61] and Control-Jujutsu [62] attacks. Consequently, the discrimination degree of AIR is not good enough. Third, *failed attacks* are conducted against a defense mechanism to demonstrate that the defense can prevent conventional attacks. However, using the same vulnerabilities, other attack methodology may be possible to break the defense. Hence, from failed attacks, we can only draw the conclusion that either the attack is flawed or the defense is effective, which does not help quantitative security evaluation much. In conclusion, we should either polish the existing metrics or design better metrics for quantitatively measuring security for CFI methods.

Chapter 6

Related Work

6.1 Control-Flow Integrity

Abadi *et al.* [1] coined the term "Control-Flow Integrity", and proposed the first implementation (ClassicCFI, detailed in §1.3.2) in 2005. Niu *et al.* [12] proposed a taxonomy that classifies CFI implementations into two categories: coarse-grained CFI and fine-grained CFI according to the equivalence class support of a particular CFI approach (details in §1.3.3).

Coarse-grained CFI approaches include PittSFIeld [8], NaCl [9, 10], CCFIR [11], binCFI [7], and MIP [12]. The major benefit of coarse-grained CFI is that coarse-grained CFGs are easier to build, even without access to source code (e.g., [13]). 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 [14–16].

Previous fine-grained CFI approaches include several systems [17–21, 25, 26, 41]. However, limited by their CFI enforcement mechanisms, none of them supports modularity. ForwardCFI [23] is a fine-grained CFI system with insecure modularity support, because its modularity support introduces time windows for attacks during dynamic module linking. Our MCFI [27] work is the first fine-grained CFI technique that supports dynamic code linking, and RockJIT [28] extends MCFI to securing Just-In-Time (JIT) compilation. The CFG generation of both ForwardCFI and MCFI (only the C++ part) is based on the idea of SafeDispatch [22]. vfGuard [80] describes a sound CFG edge generation approach at the binary level. Lockdown [81] 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.

πCFI is the first CFI approach to comprehensively enforcing per-input CFGs. Independently and concurrently with πCFI, HAFIX [82] 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. Recent attacks such as Control-Flow Bending [61] and Control Jujutsu [62] show methods of attacking conventional fine-grained CFI systems, but fortunately πCFI [29] can mitigate those attacks.

Systems such as XFI [17] 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.

Code-Pointer Integrity (CPI [83]) 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 execution overhead. However, it lacks interoperability support and incurs high memory overhead. Furthermore, CPI does not directly enforce a control-flow graph. The control-flow graph provided by CFI methods such as MCFI and πCFI is valuable to other software-protection mechanisms because they can use it to perform static-analysis based optimization and verification [25].

6.2 Software-based Fault Isolation

Software-based Fault Isolation, or SFI, was first proposed by Wahbe *et al.* [36] in 1993 on RISC to achieve efficient user-level domain isolation, and was ported to x86 in PittSFIeld [8]. For every indirect memory access, PittSFIeld masks its target address using a single `and` instruction. To ensure that no such `and` instruction is bypassed, PittSFIeld implements a coarse-grained CFI called *aligned-chunk CFI*. In this approach, the code region is divided into chunks of the same size such as 16 bytes. Branch targets are aligned at chunk beginnings. All branches are restricted to target beginnings of chunks. For indirect branches, this is achieved by dynamic checks. Thanks to the restriction on branches, instrumentation code cannot be bypassed as long as it stays in the same chunk as the protected memory access. The same approach has been adopted by NaCl [9, 10]. The downside of this approach is that many nops need to be inserted into the code for alignments, so we adopt the approach in [37] for memory sandboxing, which leverages the CPU-provided address overriding prefix.

Both CFI and SFI weave checks into the code to monitor the runtime execution, and they are essentially Inlined Reference Monitors, which are conceptualized by Erlingsson *et al.* [84, 85].

6.3 JIT Compiler Hardening

RockJIT's goal of improving the security of JIT compilation is shared by several other systems. Perhaps the closest work is NaCl-JIT [43], which applies SFI to constraining both a JIT compiler and JITted code. To prevent SFI checks from being bypassed, NaCl-JIT enforces aligned-chunk CFI similar to PittSFIeld [8], which enforces coarse-grained CFGs. In contrast, RockJIT applies fine-grained per-input CFI on the JIT compiler and therefore provides stronger security. NaCl-JIT also has high performance overhead. Its aligned-chunk CFI requires insertion of many nop instructions to make indirect-branch targets aligned at chunk boundaries. NaCl-JIT reports nops account for half of the sandboxing cost. Largely because of this, its performance overhead is around 51%. By contrast, RockJIT's overhead is 12.1%.

In addition, software diversification has been studied to harden JIT compilation. The librando system [55] inserts a random amount of nops in the JITted code. In addition, it uses a technique called constant blinding: it replaces instructions that have constant operands with other equivalent instruction sequences to mitigate JIT spraying [57]. Due to its black-box implementation, librando has to disassemble the JITted code, modify the code, and re-assemble the new code. It incurs a significant overhead (265.8%). Other systems including INSeRT [86], JITSafe [87], and RIM [88] also employ diversification techniques similar to librando's. Readactor [73] leverages execute-only pages supported by virtualization and runs randomized JIT engine and JITted code inside those pages. Most of these diversification-based systems protect only JITted code, not the JIT compiler. Even Readactor needs to temporarily allow writable code during JITted code installation. In comparison, RockJIT can eliminate JIT spraying attacks and enforces CFI on both the JIT compiler and JITted code. On the other hand, since software-diversification techniques are orthogonal to CFI, it is perhaps beneficial to deploy both defenses in a JIT compiler, following the principle of defense in depth.

Another mitigation mechanism for JIT is to separate the write permission from the execution permission for the code heap. For instance, SDCG [46] stores the shadow code heap in another process and emits code to the process through inter-process communication. However, the process-based separation seems to be heavier than RockJIT's SFI-based separation. JITDefender [89] and JITSafe [87] drop the write permission of the code heap whenever it is not needed. However, before dropping the permission, those code pages may have already been modified by the attacker for arbitrary code execution. More importantly, they cannot prevent JIT spraying attacks, which do not require modifying the code heap.

6.4 Software Transactional Memory

The idea of Transactional Memory was first proposed by Herlihy and Moss [90] in 1993, and was implemented as a simple extension to the cache coherence protocols used in multiprocessor CPUs. Later in 1995, Shavit and Touitou proposed a software-only implementation of transactional memory, coined Software Transactional Memory (STM) [91]. The original STM algorithm operates on pre-determined shared memory words and for each word, it associates an ownership record (orec) indicating the transaction that tries to modify it. A transaction needs to acquire the ownership of all words before committing its changes, and it does so in a global total order. If a transaction tries to acquire the ownership of a word that has been owned, it is aborted and retried. It has two limitations: first, it doubles memory usage by attaching an ownership record to each memory word; second, it assumes that the accessed memory words are statically known. To reduce memory overhead, Harris *et al.* proposed a Hashtable STM [92] that hashes those memory words to the same hash entry if they are owned by the same transaction. To support dynamically allocated shared memory, object-based STM systems [93, 94] were proposed. For example, DSTM [93] dynamically allocates ownership records for well-defined shared objects such as a tree node. DSTM also introduces the concept of contention manager to handle transaction conflicts more efficiently.

All of the above STM algorithms use ownership records for detecting conflicts. However, they are not suitable for MCFI's transaction design because of performance. For instance, reading object data in DSTM needs to dereference two levels of pointers, which is likely to incur large runtime overhead for CFI checking. Instead, MCFI's check transactions use version numbers to validate read consistency, whose basic idea is similar to Transactional Locking II (TL2) [95] and Transactional Mutex Locking (TML) [38]. Both TL2 and TML use a global version number and locally kept version numbers to detect read-write conflicts, but due to their generic transaction support, the version number retrieval and actual data read are separate. In MCFI, we merge the two memory reads into one single instruction using specially designed ID encoding to improve efficiency.

Chapter 7

Conclusions

In this dissertation, we have described MCFI, the first CFI system supporting fine-grained CFGs, modularity, efficiency and interoperability. MCFI adopts a source-level semantics-based method to generate fine-grained CFGs for C/C++ programs, and the generated CFG is encoded as two tables in memory during runtime, which are consulted by MCFI-inserted checks in the program for detecting CFI violations. When new code modules are loaded dynamically, the CFG tables may be concurrently updated and queried. For thread safety, MCFI uses a specially designed lightweight STM algorithm to update and query the CFG tables, which results in low performance overhead. The table and transaction design also enable interoperability.

Atop MCFI, we have proposed πCFI, a CFI technique that is able to generate and enforce per-input CFGs. πCFI inserts extra instrumentation into the target program, and 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. πCFI can effectively reduce available indirect branch edges by a large percentage, while incurring low overhead.

Finally, we presented a general approach entitled RockJIT to securing JIT compilers using πCFI. RockJIT enforces fine-grained per-input CFI on the JIT compiler and coarse-grained CFI on the JITted code, resulting in much improved security and lower performance overhead than other state-of-the-art systems.

Bibliography

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

[2] 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)*, 552–561 (2007).

[3] Shacham, H. *et al.* On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, 298–307 (ACM, New York, NY, USA, 2004). URL http://doi.acm.org/10.1145/1030083.1030124.

[4] Cowan, C. *et al.* StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, 5–5 (USENIX Association, Berkeley, CA, USA, 1998). URL http://dl.acm.org/citation.cfm?id=1267549.1267554.

[5] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D. & Boneh, D. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 227–242 (IEEE Computer Society, Washington, DC, USA, 2014). URL http://dx.doi.org/10.1109/SP.2014.22.

[6] Ramalingam, G. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* **16**, 1467–1471 (1994). URL http://doi.acm.org/10.1145/186025.186041.

[7] Zhang, M. & Sekar, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13 (2013).

[8] McCamant, S. & Morrisett, G. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06 (USENIX Association, 2006). URL http://dl.acm.org/citation.cfm?id=1267336.1267351.

[9] Yee, B. *et al.* Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Security and Privacy, 2009 IEEE Symposium on*, 79–93 (2009).

[10] Sehr, D. *et al.* Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th Usenix Security Symposium*, 1–12 (2010).

[11] Zhang, C. *et al.* Practical Control Flow Integrity and Randomization for Binary Executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, 559–573 (2013).

[12] Niu, B. & Tan, G. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 199–210 (ACM, 2013). URL http://doi.acm.org/10.1145/2508859.2516649.

[13] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W. & Franz, M. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)* (San Diego, California, 2015).

[14] Goktas, E., Athanasopoulos, E., Bos, H. & Portokalidis, G. Out Of Control: Overcoming Control-Flow Integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014).

[15] Davi, L., Lehmann, D., Sadeghi, A. & Monrose, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)* (USENIX Association, 2014).

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

[17] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M. & Necula, G. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 75–88 (2006).

[18] Wang, Z. & Jiang, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 380–395 (2010).

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

[20] Davi, L. *et al.* MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Network and Distributed System Security Symposium (NDSS)* (2012).

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

[22] Jang, D., Tatlock, Z. & Lerner, S. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *20th Annual Network and Distributed System Security Symposium*, NDSS '14 (The Internet Society, 2014).

[23] Tice, C. *et al.* Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)* (USENIX Association, 2014).

[24] Szekeres, L., Payer, M., Wei, T. & Song, D. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, 48–62 (2013).

[25] Zeng, B., Tan, G. & 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)*, 29–40 (2011).

[26] Zeng, B., Tan, G. & Erlingsson, Ú. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *22nd Usenix Security Symposium*, 369–382 (2013).

[27] Niu, B. & Tan, G. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 577–587 (ACM, 2014). URL http://doi.acm.org/10.1145/2594291.2594295.

[28] Niu, B. & 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, CCS '14, 1317–1328 (ACM, New York, NY, USA, 2014). URL http://doi.acm.org/10.1145/2660267.2660281.

[29] Niu, B. & Tan, G. Per-Input Control-Flow Integrity. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 1317–1328 (2015). URL http://dx.doi.org/10.1145/2810103.2813644.

[30] Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html.

[31] Dean, J., Grove, D. & Chambers, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, 77–101 (1995).

[32] Itanium C++ ABI. http://mentorembedded.github.io/cxx-abi/abi.html.

[33] The "`this`" pointer in c++. http://en.cppreference.com/w/cpp/language/this.

[34] C++ type qualifiers. http://en.cppreference.com/w/cpp/language/cv.

[35] Bosman, E. & Bos, H. Framing Signals - A Return to Portable Shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on*, 243–258 (2014).

[36] Wahbe, R., Lucco, S., Anderson, T. E. & Graham, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, 203–216 (ACM, 1993). URL http://doi.acm.org/10.1145/168619.168635.

[37] Deng, L., Zeng, Q. & Liu, Y. ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In Federrath, H. & Gollmann, D. (eds.) *ICT Systems Security and Privacy Protection*, vol. 455 of *IFIP Advances in Information and Communication Technology*, 386–400 (Springer International Publishing, 2015).

[38] Dalessandro, L., Dice, D., Scott, M., Shavit, N. & Spear, M. Transactional Mutex Locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, 2–13 (Springer-Verlag, Berlin, Heidelberg, 2010).

[39] Herlihy, M. P. & Wing, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* **12**, 463–492 (1990).

[40] Dechev, D. The ABA Problem in Multicore Data Structures with Collaborating Operations. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 158–167 (2011).

[41] Criswell, J., Dautenhahn, N. & Adve, V. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 292–307 (IEEE Computer Society, Washington, DC, USA, 2014). URL `http://dx.doi.org/10.1109/SP.2014.26`.

[42] Saha, S., Lozi, J.-P., Thomas, G., Lawall, J. & 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*, 1–12 (2013).

[43] Ansel, J. *et al.* Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 355–366 (2011).

[44] Section 8.1.3: Handling Self- and Cross-Modifying Code. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3.

[45] Example B2-1 Cache cleaning operations for self-modifying code. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.

[46] Song, C., Zhang, C., Wang, T., Lee, W. & 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).

[47] Snow, K. Z. *et al.* Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 574–588 (2013).

[48] Gal, A. *et al.* Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 465–478 (ACM, New York, NY, USA, 2009). URL http://doi.acm.org/10.1145/1542476.1542528.

[49] Deutsch, L. P. & Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, 297–302 (ACM, 1984). URL http://doi.acm.org.ezproxy.lib.lehigh.edu/10.1145/800017.800542.

[50] Hölzle, U., Chambers, C. & Ungar, D. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In America, P. (ed.) *ECOOP'91 European Conference on Object-Oriented Programming*, vol. 512 of *Lecture Notes in Computer Science*, 21–38 (Springer Berlin Heidelberg, 1991). URL http://dx.doi.org/10.1007/BFb0057013.

[51] Hölzle, U., Chambers, C. & Ungar, D. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, 32–43 (ACM, 1992).

[52] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J. & Gan, E. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 395–404 (ACM, 2012). URL http://doi.acm.org/10.1145/2254064.2254111.

[53] Fredkin, E. Trie Memory. *Communications of ACM* **3**, 490–499 (1960).

[54] Seaborn, M. A dfa-based x86-32 validator for native client. https://github.com/mseaborn/x86-decoder (2011).

[55] Homescu, A., Brunthaler, S., Larsen, P. & Franz, M. Librando: Transparent Code Randomization for Just-In-Time Compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, CCS '13, 993–1004 (ACM, 2013). URL http://doi.acm.org/10.1145/2508859.2516675.

[56] The rp++ ROP gadget finding tool. https://github.com/0vercl0k/rp.

[57] Blazakis, D. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, 1–9 (USENIX Association, 2010). URL http://dl.acm.org/citation.cfm?id=1925004.1925011.

[58] Sintsov, A. Safari JS JITed Shellcode. http://www.exploit-db.com/exploits/14221/ (2010).

[59] Badishi, G. JIT Spraying Primer and CVE-2010-3654. http://badishi.com/jit-spraying-primer-and-cve-2010-3654/ (2012).

[60] Schuster, F. *et al.* Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (2015).

[61] Carlini, N., Barresi, A., Payer, M., Wagner, D. & Gross, T. R. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)* (USENIX Association, Washington, D.C., 2015).

[62] Evans, I. *et al.* Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (2015).

[63] Chen, S., Xu, J., Sezer, E. C., Gauriar, P. & Iyer, R. K. Non-control-data Attacks Are Realistic Threats. In *In USENIX Security Symposium*, 177–192 (2005).

[64] HeartBleed. http://heartbleed.com/.

[65] Hu, H., Chua, Z. L., Adrian, S., Saxena, P. & Liang, Z. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, 177–192 (USENIX Association, Washington, D.C., 2015).

[66] Apple Goto Fail. https://gotofail.com/.

[67] Castro, M., Costa, M. & Harris, T. Securing Software by Enforcing Data-flow Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 147–160 (2006).

[68] Nagarakatte, S., Zhao, J., Martin, M. M. K. & Zdancewic, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 245–258 (2009).

[69] Akritidis, P., Costa, M., Castro, M. & Hand, S. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *18th Usenix Security Symposium*, 51–66 (2009).

[70] Nagarakatte, S., Zhao, J., Martin, M. M. & Zdancewic, S. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, 31–40 (ACM, New York, NY, USA, 2010). URL http://doi.acm.org/10.1145/1806651.1806657.

[71] Serebryany, K., Bruening, D., Potapenko, A. & Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 28–28 (USENIX Association, Berkeley, CA, USA, 2012). URL http://dl.acm.org/citation.cfm?id=2342821.2342849.

[72] Backes, M. *et al.* You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 1342–1353 (ACM, New York, NY, USA, 2014). URL http://doi.acm.org/10.1145/2660267.2660378.

[73] Crane, S. *et al.* Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, 763–780 (2015).

[74] Crane, S. J. *et al.* It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 243–255 (ACM, New York, NY, USA, 2015). URL http://doi.acm.org/10.1145/2810103.2813682.

[75] Tang, A., Sethumadhavan, S. & Stolfo, S. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *Proceedings of the 22Nd ACM SIGSAC Conference on*

Computer and Communications Security, CCS '15, 256–267 (ACM, New York, NY, USA, 2015). URL http://doi.acm.org/10.1145/2810103.2813685.

[76] Bigelow, D., Hobson, T., Rudd, R., Streilein, W. & Okhravi, H. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 268–279 (ACM, New York, NY, USA, 2015). URL http://doi.acm.org/10.1145/2810103.2813691.

[77] Lu, K. *et al.* ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 280–291 (ACM, New York, NY, USA, 2015). URL http://doi.acm.org/10.1145/2810103.2813694.

[78] Klein, G. *et al.* seL4: Formal Verification of An OS Kernel. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 207–220 (2009).

[79] Schwartz, E. J., Avgerinos, T. & Brumley, D. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, 25–25 (USENIX Association, Berkeley, CA, USA, 2011). URL http://dl.acm.org/citation.cfm?id=2028067.2028092.

[80] Prakash, A., Hu, X. & Yin, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014* (2015).

[81] Payer, M., Barresi, A. & 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, 2015).

[82] Arias, O. *et al.* HAFIX: Hardware-Assisted Flow Integrity Extension. In *52nd Design Automation Conference (DAC)* (2015).

[83] Kuznetsov, V. *et al.* Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 147–163 (2014).

[84] Erlingsson, Ú. & Schneider, F. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, 87–95 (ACM Press, 1999).

[85] Erlingsson, Ú. & Schneider, F. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy (S&P)*, 246–255 (2000).

[86] Wei, T., Wang, T., Duan, L. & Luo, J. INSeRT: Protect Dynamic Code Generation against Spraying. In *Information Science and Technology (ICIST), 2011 International Conference on*, 323–328 (2011).

[87] Chen, P., Wu, R. & Mao, B. JITSafe: A Framework against Just-In-Time Spraying Attacks. *Information Security, IET* **7**, 283–292 (2013).

[88] Wu, R., Chen, P., Mao, B. & Xie, L. RIM: A Method to Defend from JIT Spraying Attack. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, 143–148 (2012).

[89] Chen, P., Fang, Y., Mao, B. & Xie, L. JITDefender: A Defense against JIT Spraying Attacks. In *26th IFIP International Information Security Conference*, vol. 354, 142–153 (2011).

[90] Herlihy, M. & Moss, J. E. B. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, 289–300 (ACM, New York, NY, USA, 1993). URL http://doi.acm.org/10.1145/165123.165164.

[91] Shavit, N. & Touitou, D. Software Transactional Memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, 204–213 (1995).

[92] Harris, T. & Fraser, K. Language Support for Lightweight Transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, 388–402 (ACM, New York, NY, USA, 2003).

[93] Herlihy, M., Luchangco, V., Moir, M. & Scherer, W. N., III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium*

on Principles of Distributed Computing, PODC '03, 92–101 (ACM, New York, NY, USA, 2003). URL http://doi.acm.org/10.1145/872035.872048.

[94] Fraser, K. *Practical Lock-Freedom*. Ph.D. thesis, University of Cambridge (2003).

[95] Dice, D., Shalev, O. & Shavit, N. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, 194–208 (Springer-Verlag, Berlin, Heidelberg, 2006). URL http://dx.doi.org/10.1007/11864219_14.

Vita

Ben Niu was born in Lishi, a small city in Shanxi province, China in June, 1988. His parents, Siping Niu and Tuqin Qiao, are both civil engineers. He spent fourteen years in Lishi and finished 8th grade in No.1 Junior High School. Shortly, he moved to the city of Jinzhong and attended Xinxing Senior High School, where he had three wonderful years of study. After graduation from Xinxing, he enrolled at Zhejiang University in Hangzhou, a fabulous city in East China, majoring in Software Engineering. There, he was fascinated by computer systems and security, which later became his major research interests. In 2009, he graduated from Zhejiang University with honors and enrolled in its graduate school. One year later, he dropped out and came to Lehigh University in the U.S., pursuing a doctoral degree in the area of computer system security. He has been a co-author of a few papers published at prestigious computer science venues such as ACM PLDI and CCS. He has also been an external reviewer for several conferences. He was a software engineering intern at FireEye in the summer of 2014 and a security software engineering intern at Microsoft in summer 2015.

Publications

* *Per-Input Control-Flow Integrity*. Ben Niu and Gang Tan. At the twenty- second ACM Conference on Computer and Communications Security (CCS), 2015

* *RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity*. Ben Niu and Gang Tan. At the twenty-first ACM Conference on Computer and Communications Security (CCS), 2014.

* *Modular Control-Flow Integrity*. Ben Niu and Gang Tan. At the thirty-fifth annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 2014.

* *Monitor Integrity Protection with Space Efficiency and Separate Compilation*. Ben Niu and Gang Tan. At the twentieth ACM Conference on Computer and Communications Security (CCS), 2013.

* *Efficient User-Space Information Flow Control*. Ben Niu and Gang Tan. In the eighth ACM Symposium on Information, Computer and Communication Security (AsiaCCS), 2013.

* *Enforcing User-Space Privilege Separation with Declarative Architectures*. Ben Niu and Gang Tan. In the seventh ACM Workshop on Scalable Trusted Computing (STC), 2012.