Control-Flow Bending: On the Effectiveness of Control-Flow Integrity

Nicolas Carlini
UC Berkeley

Antonio Barresi
ETH Zurich

Mathias Payer
Purdue University

David Wagner
UC Berkeley

Thomas R. Gross
ETH Zurich

Abstract

Control-Flow Integrity (CFI) is a defense which prevents control-flow hijacking attacks. While recent research has shown that coarse-grained CFI does not stop attacks, fine-grained CFI is believed to be secure.

We argue that assessing the effectiveness of practical CFI implementations is non-trivial and that common evaluation metrics fail to do so. We then evaluate fully-precise static CFI — the most restrictive CFI policy that does not break functionality — and reveal limitations in its security. Using a generalization of non-control-data attacks which we call Control-Flow Bending (CFB), we show how an attacker can leverage a memory corruption vulnerability to achieve Turing-complete computation on memory using just calls to the standard library. We use this attack technique to evaluate fully-precise static CFI on six real binaries and show that in five out of six cases, powerful attacks are still possible. Our results suggest that CFI may not be a reliable defense against memory corruption vulnerabilities.

We further evaluate shadow stacks in combination with CFI and find that their presence for security is necessary: deploying shadow stacks removes arbitrary code execution capabilities of attackers in three of six cases.

1 Introduction

Attacking software systems by exploiting memory-corruption vulnerabilities is one of the most common attack methods today according to the list of Common Vulnerabilities and Exposures. To counter these threats, several hardening techniques have been widely adopted, including ASLR [29], DEP [38], and stack canaries [10]. Each has limitations: stack canaries protect only against contiguous overwrites of the stack, DEP protects against code injection but not against code reuse, and ASLR does not protect against information leakage.

We classify defense mechanisms into two broad categories: *prevent-the-corruption* and *prevent-the-exploit*. Defenses that prevent the corruption stop the actual memory corruption before it can do any harm to the program (i.e., no attacker-controlled values are ever used out-of-context). Examples for prevent-the-corruption defenses are SoftBound [22], Data-Flow Integrity [6], or Code-Pointer Integrity [18]. In contrast, prevent-the-exploit defenses allow memory corruption to occur but protect the application from subsequent exploitation; they try to survive or tolerate adversarial corruption of memory. Examples for prevent-the-exploit defenses are DEP [38] or stack canaries [10].

Control-Flow Integrity (CFI) [1, 3, 12, 15, 27, 30, 31, 39, 41–44] is a promising stateless prevent-the-exploit defense mechanism that aims for complete protection against control-flow hijacking attacks under a threat model with a powerful attacker that can read and write into the process's address space. CFI ensures that program execution follows a valid path through the static Control-Flow Graph (CFG). Any deviation from the CFG is a CFI violation, terminating the application. CFI is not specific to any particular exploitation vector for control-flow hijacking. Rather, it enforces its policy on all indirect branch instructions. Therefore any attempt by an attacker to alter the control-flow in an invalid manner will be detected, regardless of *how* the attacker changes the target of the control-flow transfer instruction.

CFI is often coupled with a protected shadow stack, which ensures that each return statement matches the corresponding call and thereby prevents an attacker from tampering with return addresses. While the foundational work [1,15] included a shadow stack as part of CFI, some more recent research has explored variants of CFI that omit the shadow stack for better performance. Whereas conformance to the CFG is a stateless policy, shadow stacks are inherently dynamic and are more precise than any static policy can be with respect to returns.

Many prior attacks on CFI have focused on attacking a weak or suboptimal implementation of CFI. Our focus is on evaluating the effectiveness of CFI in its best achiev-

able form, instead of artifacts of some (possibly weak) CFI implementation. We define *fully-precise static CFI* as the best achievable CFI policy as follows: a branch from one instruction to another is allowed if and only if some benign execution makes that same control-flow transfer. Such a policy could be imagined as taking any CFG over-approximation and removing edges until removing additional edges would break functionality.

Thus, fully-precise static CFI is the most restrictive stateless CFI policy that still allows the program to run as intended. Both coarse-grained and fine-grained CFI are less precise than fully-precise static CFI, because they both over-approximate the set of valid targets for each indirect transfer (though to a different degree). In contrast, fully-precise static CFI involves no approximation by definition. We acknowledge that fully-precise static CFI might be stricter than anything that can be practically implemented, but this makes any attacks all the more meaningful: our results help us understand fundamental limits on the effectiveness of the strongest possible CFI policy.

Through several methods of evaluation, we argue that fully-precise static CFI is neither completely broken (as most coarse-grained defenses are) nor totally secure. We explore what CFI can and cannot prevent, and hope that this will stimulate a broader discussion about ways to further strengthen CFI.

We evaluate the security of fully-precise static CFI both with and without shadow stacks. Recent research achieves better performance by omitting the shadow stack in favor of a static policy on return statements. We still call it fully-precise static CFI when we have added a shadow stack, because the shadow stack is orthogonal. This does not change the fact that the CFI policy is static.

CFI works by preventing an attacker from deviating from the control-flow graph. Our attacks do not involve breaking the CFI mechanism itself: we even assume the mechanism is implemented perfectly to its fullest extent. Rather, our analysis demonstrates that an attacker can still create exploits for most real applications, without causing execution to deviate from the control-flow graph.

This paper provides the following contributions:

1. formalization and evaluation of a space of different kinds of CFI schemes;
2. new attacks on fully-precise static CFI, which reveal fundamental limits on the effectiveness of CFI;
3. evidence that existing metrics for CFI security are ineffective;
4. evidence that CFI without a shadow stack is broken;
5. widely applicable Turing-complete attacks on CFI with shadow stacks; and,
6. practical case studies of the security of fully-precise static CFI for several existing applications.

2 Background and software attacks

Over the past few decades, one of the most common attack vectors has been exploitation of memory corruption within programs written in memory-unsafe languages.In response, operating systems and compilers have started to support countermeasures against specific exploitation vectors and vulnerability types, but current hardening techniques are still unable to stop all attacks. We briefly provide an overview of these attacks; more information may be found elsewhere [37].

2.1 Control-Flow Hijacking

One way to exploit a memory corruption bug involves hijacking control flow to execute attacker-supplied or already-existing code in an application's address space. These methods leverage the memory corruption bug to change the target of an indirect branch instruction (*ret*, *jmp* *, or *call* *). By doing so, an attacker can completely control the next instructions to execute.

2.2 Code-Reuse Attacks

Data Execution Prevention (DEP) prevents executing attacker-injected code. However, redirecting control-flow to already-existing executable code in memory remains feasible. One technique, return-to-libc [25, 36], reuses existing functions in the address space of the vulnerable process. Runtime libraries (such as libc) often provide powerful functions, e.g., wrapper functions for most system calls. One example is libc's `system()` function, which allows the attacker to execute shell commands. Code-reuse attacks are possible when attacker-needed code is already available in the address space of a vulnerable process.

2.3 Return Oriented Programming

Return Oriented Programming (ROP) [25, 36] is a more advanced form of code-reuse attack that lets the attacker perform arbitrary computation solely by reusing existing code. It relies upon short instruction sequences (called "gadgets") that end with an indirect branch instruction. This allows them to be chained, so the attacker can perform arbitrary computation by executing a carefully-chosen sequence of gadgets. ROP can be generalized to use indirect jump or call instructions instead of returns [4, 7].

2.4 Non-Control-Data Attacks

A non-control-data attack [8] is an attack where a memory corruption vulnerability is used to corrupt only data,

but not any code pointer. (A *code pointer* is a pointer which refers to the code segment, for example, a return address or function pointer.) Depending on the circumstances, these attacks can be as effective as arbitrary code-execution attacks. For instance, corrupting the parameter to a sensitive function (e.g., libc's `execve()`) may allow an attacker to execute arbitrary programs. An attacker may also be able to overwrite security configuration values and disable security checks. Non-control-data attacks are realistic threats and hard to defend against, due to the fact that most defense mechanisms focus on the protection of code pointers.

2.5 Control-Flow Bending

We introduce a generalization of non-control-data attacks which we call *Control-Flow Bending* (CFB). While non-control-data attacks do not directly modify any control-flow data (e.g., return addresses, indirect branch targets), in control-flow bending we allow these modifications so long as the modified indirect branch target is still in the valid set of addresses as defined by the CFI policy (or any other enforced control-flow or code pointer integrity protection). CFB allows an attacker to bend the control-flow of the application (compared to hijacking it) but adheres to an imposed security policy.

We define a "data-only" attack as a non-control-data attack where the entire execution trace is identical to some feasible non-exploit execution trace. (An execution trace is the ordered sequence of instructions which execute, and does not include the effects those instructions have except with respect to control flow.) While data-only attacks may change the control flow of an application, the traces will still look legitimate, as the observed trace can also occur during valid execution. In contrast, CFB is more general: it refers to any attack where each control-flow transfer is within the valid CFG, but the execution trace is not necessarily required to match some valid non-exploit trace.

In general, defence mechanisms implement an abstract machine and can only observe security violations according to the restrictions of that machine, e.g., CFI enforces that control flow follows a finite state machine.

For example, an attacker who directly overwrites the arguments to `exec()` is performing a data-only attack: no control flow has been changed. An attacker who overwrites an `is_admin` flag half-way through processing a request is performing a non-control-data attack: the data that was overwritten is non-control-data, but it affects the control-flow of the program. An attacker who modifies a function pointer to point to a different (valid) call target is mounting a CFB attack.

3 Threat model and attacker goals

Threat model. For this paper we assume a powerful yet realistic threat model. We assume the attacker can write arbitrarily to memory at one point in time during the execution of the program. We assume the process is running with non-executable data and non-writeable code which is hardware enforced.

This threat model is a realistic generalization of memory corruption vulnerabilities: the vulnerability typically gives the attacker some control over memory. In practice there may be a set of specific constraints on what the attacker can write where; however, this is not something a defender can rely upon. To be a robust defense, CFI mechanisms must be able to cope with arbitrary memory corruptions, so in our threat model we allow the attacker full control over memory once.

Limiting the memory corruption to a single point in time does weaken the attacker. However, this makes our attacks all the more meaningful.

Attacker goals. There are three kinds of outcomes an attacker might seek, when exploiting a vulnerability:

1. *Arbitrary code execution:* The attacker can execute arbitrary code and can invoke arbitrary system calls with arbitrary parameters. In other words, the attacker can exercise all permissions that the application has. Code execution might involve injecting new code or re-using already-existing code; from the attacker's perspective, there is no difference as long as the effects are the same.

2. *Confined code execution:* The attacker can execute arbitrary code within the application's address space, but cannot invoke arbitrary system calls. The attacker might be able to invoke a limited set of system calls (e.g., the ones the program would usually execute, or just enough to send information back to the attacker) but cannot exercise all of the application's permissions. Reading and leaking arbitrary memory of the vulnerable program is still possible.

3. *Information leakage:* The attacker can read and leak arbitrary values from memory.

Ideally, a CFI defense would prevent all three attacker goals. The more it can prevent, the stronger the defense.

4 Definition of CFI flavors

Control-Flow Integrity (CFI) [1, 15] adds a stateless check before each indirect control-flow transfer (indirect jump/call, or function return) to ensure that the target location is in a static set defined by the control-flow graph.

4.1 Fully-Precise Static CFI

We define *Fully-Precise Static CFI* as follows: an indirect control-flow transfer along some edge is allowed only if there exists a non-malicious trace that follows that edge. (An execution is not malicious if it exercises only intended program behavior.) In other words, consider the most restrictive control-flow graph that still allows all feasible non-malicious executions, i.e., the CFG contains an edge if and only if that edge is used by some benign execution. Fully-precise static CFI then enforces that execution follows this CFG. Thus, fully-precise static CFI enforces the most precise (and most restrictive) policy possible that does not break funcionality.

We know of no way to implement fully-precise static CFI: real implementations often use static analysis and over-approximate the CFG and thus are not fully precise. We do not design a better CFI scheme. The goal of our work is to evaluate the strongest form of CFI that could conceptually exist, and attempt to gain insight on its limitations. This notion of fully-precise static CFI allows us to transcend the recent arms race caused by defenders proposing forms of CFI [9, 28] and then attackers defeating them [5, 14, 16].

4.2 Practical CFI

Practical implementations of CFI are always limited by the precision of the CFG that can be obtained. Current CFI implementations face two sources of over-approximation. First, due to challenges in accurate static analysis, the set of allowed targets for each indirect call instruction typically depends only upon the function pointer type, and this set is often larger than necessary.

Second, most CFI mechanisms use a static points-to analysis to define the set of allowed targets for each indirect control transfer. Due to imprecisions and limitations of the analysis (e.g., aliasing in the case of points-to analysis) several sets may be merged, leading to an over-approximation of allowed targets for individual indirect control-flow transfers. The degree of over-approximation affects the precision and effectiveness of practical CFI mechanisms.

Previous work has classified practical CFI defenses into two categories: coarse-grained and fine-grained. Intuitively, a defense is fine-grained if it is a close approximation of fully-precise static CFI and coarse-grained if there are many unnecessary edges in the sets.

4.3 Stack integrity

The seminal work on CFI [1] combined two mechanisms: restricting indirect control transfers to the CFG, and a shadow call stack to restrict return instructions. The shadow stack keeps track of the current functions on the application call stack, storing the return instruction pointers in a separate region that the attacker cannot access. Each return instruction is then instrumented so that it can only return to the function that called it. For compatibility with exceptions, practical implementations often allow return instructions to return to any function on the shadow stack, not just the one on the top of the stack. As a result, when a protected shadow stack is in use, the attacker has very limited influence over return instructions: all the attacker can do is unwind stack frames. The attacker cannot cause return instructions to return to arbitrary other locations (e.g., other call-sites) in the code.

Unfortunately, a shadow stack does introduce performance overhead, so some modern schemes have proposed omitting the shadow stack. We analyze both the security of CFI with a shadow stack and CFI without a shadow stack. We assume the shadow stack is protected somehow and cannot be overwritten; we do not consider attacks against the implementation of the shadow stack.

5 Evaluating practical CFI

While there has been considerable research on how to make CFI more fine-grained and efficient, most CFI publications still lack a thorough security evaluation. In fact, the security evaluation is often limited to coarse metrics such as Average Indirect target Reduction (AIR) or gadget reduction. Evaluating the security effectiveness of CFI this way does not answer how effective these policies are in preventing actual attacks.

In this section, we show that metrics such as AIR and gadget reduction are not good indicators for the effectiveness of a CFI policy, even for simple programs. We discuss CFI effectiveness and why it is difficult to measure with a single value and propose a simple test that indicates if a CFI policy is trivially broken.

5.1 AIR and gadget reduction

The AIR metric [44] measures the relative reduction in the average number of valid targets for all indirect branch instructions that a CFI scheme provides: without CFI, an indirect branch could target any instruction in the program; CFI limits this to a set of valid targets. The gadget reduction metric measures the relative reduction in the number of gadgets that can be found at locations that are valid targets for an indirect branch instruction.

These metrics measure how effectively a CFI implementation reduces the set of valid targets (or gadgets) for indirect branch instructions, *on average*. However, they fail to capture both (i) the target reduction of individual locations (e.g., a scheme can have high AIR even if one

branch instruction has a large set of surplus targets, if the other locations are close to optimal) and (ii) the importance and risk of the allowed control transfers. Similarly, the gadget reduction metric does not weight targets according to their usefulness to an attacker: every code location or gadget is considered to be equally useful.

For example, consider an application with 10MB of executable memory and an AIR of 99%. An attacker would still have 1% of the executable memory at their disposal — 100,000 potential targets — to perform code-reuse attacks. A successful ROP attack requires only a handful of gadgets within these potential targets, and empirically, 100,000 targets is much more than is usually needed to find those gadgets [35]. As this illustrates, averages and metrics that are relative to the code size can be misleading. What is relevant is the absolute number of available gadgets and how useful they are to an attacker.

5.2 CFI security effectiveness

Unfortunately, it is not clear how to construct a single metric that accurately measures the effectiveness of CFI. Ideally, we would like to measure the ability of CFI to stop an attacker from mounting a control-flow hijack attack. More specifically, a CFI effectiveness metric should indicate whether control-flow hijacking and code-reuse attacks are still possible under a certain attacker model or not, and if so, how much harder it is for an attacker to perform a successful attack in the presence of CFI. However, what counts as successful exploitation of a software vulnerability depends on the goals of the attacker (see Section 3) and is not easily captured by a single number.

These observations suggest that assessing CFI effectiveness is hard, especially if no assumptions are made regarding what a successful attack is and what the binary image of the vulnerable program looks like.

5.3 Basic exploitation test

We propose a *Basic Exploitation Test* (BET): a simple test to quickly rule out some trivially broken implementations of CFI. Passing the BET is not a security guarantee, but failing the BET means that the CFI scheme is insecure.

In particular, the BET involves selecting a minimal program — a simple yet representative program that contains a realistic vulnerability — and then determining whether attacks are still possible if that minimal program is protected by the CFI scheme under evaluation. The minimal program should be chosen to use a subset of common run-time libraries normally found in real applications, and constructed so it contains a vulnerability that allows hijacking control flow in a way that is seen in real-life attacks. For instance, the minimal program might allow an attacker to overwrite a return address or the target of an indirect jump/call instruction.

The evaluator then applies the CFI scheme to the minimal program, selects an attacker goal from Section 3, and determines whether that goal is achievable on the protected program. If the attack is possible, the CFI scheme fails the BET. We argue that if a CFI scheme is unable to protect a minimal program it will also fail to protect larger real-life applications, as larger programs afford the attacker even more opportunities than are found in the minimal program.

5.4 BET for coarse-grained CFI

We apply the BET to a representative coarse-grained CFI policy. We show that the scheme is broken, even though its AIR and gadget reduction metrics are high. This demonstrates that AIR and gadget reduction numbers are not reliable indicators for the security effectiveness of a CFI scheme even for small programs. These results generalize the conclusion of recent work [5,14,16], by showing that coarse-grained CFI schemes are broken even for trivially small real-life applications.

Minimal program and attacker goals. Our minimal vulnerable program is shown in Figure 1. It is written in C, compiled with gcc version 4.6.3 under Ubuntu LTS 12.04 for x86 32-bit, and dynamically linked against `ld-linux` and `libc`. The program contains a stack-based buffer overflow. A vulnerability in `vulnFunc()` allows an attacker to hijack the return target of `vulnFunc()` and a memory leak in `memLeak()` allows the attacker to bypass stack canaries and ASLR.

Coarse-grained CFI policy. The coarse-grained CFI policy we analyze is a more precise version of several recently proposed static CFI schemes [43, 44]: each implementation is less accurate than our combined version. We use a similar combined static CFI policy as used in recent work [14, 16].

Our coarse-grained CFI policy has three equivalence classes, one for each indirect branch type. Returns and indirect jumps can target any instruction following a call instruction. Indirect calls can target any defined symbol, i.e., the potential start of any function. This policy is overly strict, especially for indirect jumps; attacking a stricter coarse-grained policy makes our results stronger.

Results. We see in Table 1 that our minimal program linked against its libraries achieves high AIR and gadget reduction numbers for our coarse-grained CFI policy. However, as we will show, all attacker goals from Section 3 can be achieved.

```c
#include <stdio.h>
#include <string.h>
#define STDIN 0

void memLeak() {
    char buf[64];
    int nr, i;
    unsigned int *value;
    value = (unsigned int*)buf;
    scanf("%d", &nr);
    for (i = 0; i < nr; i++)
        printf("0x%08x␣", value[i]);
}

void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}

int main(int argc, char* argv[]) {
    setbuf(stdout, NULL);
    printf("echo>␣");
    memLeak();
    printf("\nread>␣");
    vulnFunc();
    printf("\ndone.\n");
    return 0;
}
```

Figure 1: Our minimal vulnerable program that allows hijacking a return instruction target.

	AIR	Gadget red.	Targets	Gadgets
No CFI	0%	0%	1850580	128929
CFI	99.06%	98.86%	19611	1462

Table 1: Basic metrics for the minimal vulnerable program under no CFI and our coarse-grained CFI policy.

We first identified all gadgets that can be reached without violating the given CFI policy. We found five gadgets that allow us to implement all attacker goals as defined in Section 3. All five gadgets were within `libc` and began immediately following a call instruction. Two gadgets can be used to load a set of general purpose registers from the attacker-controlled stack and then return. One gadget implements an arbitrary memory write ("write-what-where") and then returns. Another gadget implements an arbitrary memory read and then returns. Finally, we found a fifth gadget — a "call gadget" — that ends with an indirect call through one of the attacker-controlled registers, and thus can be used to perform arbitrary calls. The five gadgets are shown in Figure 2. By routing control-flow through the first four gadgets and then to the call gadget, the attacker can call any function.

The attacker can use these gadgets to execute arbitrary system calls by calling `__kernel_vsyscall`. In Linux systems (x86 32-bit), system calls are routed through a virtual dynamic shared object (`linux-gate.so`) mapped into user space by the kernel at a random address. The address is passed to the user space process. If the address is leaked, the attacker can execute arbitrary system calls by calling `__kernel_vsyscall` using a call gadget. Calls to `__kernel_vsyscall` are within the allowed call targets as `libc` itself calls `__kernel_vsyscall`.

```
G1 # arbitrary load (1/2)
   f38ff:       pop    %edx
   f3900:       pop    %ecx
   f3901:       pop    %eax
   f3902:       ret

G2 # arbitrary load (2/2)
   412d2:       add    $0x20,%esp
   412d5:       xor    %eax,%eax
   412d7:       pop    %ebx
   412d8:       pop    %esi
   412d9:       pop    %edi
   412da:       ret

G3 # arbitrary read
   2ee25:       add    $0x1771cf,%ecx
   2ee2b:       mov    0x54(%ecx),%eax
   2ee31:       ret

G4 # arbitrary write
   3fb11:       pop    %ecx
   3fb12:       add    $0xa,%ecx
   3fb18:       mov    %ecx,(%edx)
   3fb1a:       ret

G5 # arbitrary call
   1b008:       mov    %esi,(%esp)
   1b00b:       call   *%edi
```

Figure 2: Our call-site gadgets within libc.

```
000b8d60 <execve>:
...
   b8d72:       call   ...
   b8d77:       add    $0xed27d,%ebx
   b8d7d:       mov    0xc(%esp),%edi
   b8d81:       xchg   %ebx,%edi
   b8d83:       mov    $0xb,%eax
   b8d88:       call   *%gs:0x10
```

Figure 3: Disassembly of `libc`'s `execve()` function. There is an instruction (`0xb8d77`) that can be returned to by any return gadget under coarse-grained CFI.

Alternatively, the attacker could call `libc`'s wrappers for each specific system call. For example, the attacker could call `execve()` within `libc` to execute the execve system call. Interestingly, if the wrapper functions contain calls, we can directly return to an instruction after such a call and before the system call is issued. For an example, see Figure 3: returning to `0xb8d77` allows us to directly issue the system call without using the call gadget (we simply direct one of the other gadgets to return there). There are some side effects on register `ebx` and `edi` but it is straightforward to take them into account.

Arbitrary code execution is also possible. In the absence of CFI, an attacker might write new code somewhere into memory, call `mprotect()` to make that memory region executable, and then jump to that location

to execute the injected code. CFI will prevent this, as the location of the injected code will never be in one of the target sets. We bypass this protection by using `mprotect()` to make already-mapped code writeable. The attacker can overwrite these already-available code pages with malicious code and then transfer control to it using our call gadget. The result is that the attacker can inject and execute arbitrary code and invoke arbitrary system calls with arbitrary parameters. As an alternative `mmap()` could also be used to allocate readable and executable memory (if not prohibited).

The minimal program shown in Figure 1 contains a vulnerability that allows the attacker to overwrite a return address on the stack. We also analyzed other minimal programs that allow the attacker to hijack an indirect jump or indirect call instruction, with similar results. We omit the details of these analyses for brevity. A minimal vulnerable program for initial indirect jump or indirect call hijacking is found in Appendix A.

Based on these results we conclude that coarse-grained CFI policies are not effective in protecting even small and simple programs, such as our minimal vulnerable program example. Our analysis also shows that AIR and gadget reduction metrics fail to indicate whether a CFI scheme is effective at preventing attacks; if such attacks are possible on a small program, then attacks will be easier on larger programs where the absolute number of valid locations and gadgets is even higher.

6 Attacks on Fully-Precise Static CFI

We now turn to evaluating fully-precise static CFI. Recall from Section 2.5 that we define control-flow bending as a generalization of non-control-data attacks. We examine the potential for control-flow bending attacks on CFI schemes with and without a shadow stack.

6.1 Necessity of a shadow stack

To begin, we argue that CFI must have a shadow stack to be a strong defense. Without one, an attacker can easily traverse the CFG to reach almost any program location desired and thereby break the CFI scheme.

For a static, stateless policy like fully-precise static CFI without a shadow stack, the best possible policy for returns is to allow return instructions within a function F to target any instruction that follows a call to F. However, for functions that are called often, this set can be very large. For example, the number of possible targets for the return statements in `malloc()` is immense. Even though dynamically only one of these should be allowed at any given time, a stateless policy must allow all of these edges.

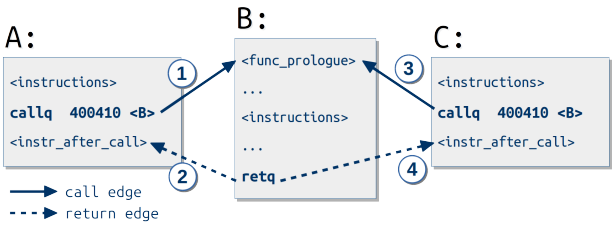

Figure 4: A control-flow graph where the lack of a shadow stack allows an attacker to mount a control-flow bending attack.

This is elaborated in Figure 4. Functions A and C both contain calls to function B. The return in function B must therefore be able to target the instruction following both of these calls. In normal execution, the program will execute edge 1 followed by edge 2, or edge 3 followed by edge 4. However, an attacker may be able to cause edge 3 to be followed by edge 2, or edge 1 to be followed by edge 4.

In practice this is even more problematic with tail-call optimizations, when signal handlers are used, or when the program calls `setjmp`/`longjmp`. We ignore these cases. This makes our job as an attacker more difficult, but we base our attacks on the fundamental properties of CFI instead of corner cases which might be handled separately.

6.1.1 Dispatcher functions

For an attacker to cause a function to return to a different location than it was called from, she must be able to overwrite the return address on the stack after the function is called yet before it returns. This is easy to arrange when the memory corruption vulnerability occurs within that specific function. However, often the vulnerability is found in uncommonly called (not well tested) functions.

To achieve more power, we make use of *dispatcher functions* (analogous to dispatcher gadgets for JOP [4]). A dispatcher function is one that can overwrite its own return address when given arguments supplied by an attacker. If we can find a dispatcher function that will be called later and use the vulnerability to control its arguments, we can make it overwrite its own return address. This lets us return to any location where this function was called.

Any function that contains a "write-what-where" primitive when the arguments are under the attacker's control can be used as a dispatcher function. Alternatively, a function that can write to only limited addresses can still work as long as the return address is within the limits. Not every function has this property, but a significant fraction of all functions do. For example, assume we control all of the arguments to `memcpy()`. We can

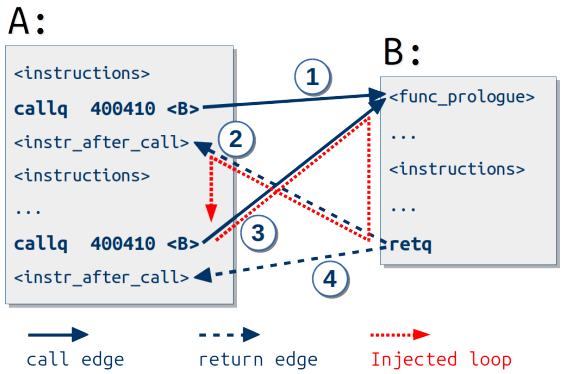

Figure 5: An example of loop injection. Execution follows call edge ③, then returns along edge ②.

point the source buffer to an attacker-controlled location, the target buffer to the address where `memcpy()`'s return address will be found, and set the length to the word size. Then, when `memcpy()` is invoked, `memcpy()` will overwrite its own return address and then return to some other location in the code chosen by the attacker. If this other location is in the valid CFG (i.e., it is an instruction following some call to `memcpy()`), then it is an allowed edge and CFI will allow the return. Thus, `memcpy()` is a simple example of a dispatcher function.

We found many dispatcher functions in `libc`, e.g.,

1. `memcpy()` — As described above.
2. `printf()` — Using the "%n" format specifier, the attacker can write an arbitrary value to an arbitrary location and thus cause `printf()` to overwrite its own return address.
3. `strcat()` — Similar to `memcpy()`. Only works if the address to return to does not contain null bytes.
4. `fputs()` — We rely on the fact that when `fputs()` is called, characters are first temporarily buffered to a location as specified in the `FILE` argument. An attacker can therefore specify a `FILE` where the temporary buffer is placed on top of the return address. Most functions that take a `FILE` struct as an argument can be used in a similar manner.

Similar functions also exist in Windows libraries. Application-specific dispatcher functions can be useful as well, as they may be called more often.

Any function that calls a dispatcher function is itself a dispatcher function: instead of having the callee overwrite its own address, it can be used to overwrite the return address of its caller (or higher on the call chain).

6.1.2 Loop injection

One further potential use of dispatcher functions is that they can be used to create loops in the control-flow graph when none were intended, a process which we call *loop injection*. We can use this to help us achieve Turing-complete computation if we require a loop.

Consider the case where there are two calls to the same dispatcher function, where the attacker controls the arguments to the second call and it is possible to reach the second call from the first through a valid CFG path. For example, it is common for programs to make multiple successive calls to `printf()`. If the second call to `printf()` allows an attacker to control the arguments, then this could cause a potential loop. This is achievable because the second call to `printf()` can return to the instruction following the first call to `printf()`. We can then reach the second call to `printf()` from there (by assumption) and we have completed the loop.

Figure 5 contains an example of this case. Under normal execution, function A would begin by executing the first call to function B on edge 1. Function B returns on edge 2, after which function A continues executing. The second call to function B is then executed, on edge 3. B this time returns on edge 4. Notice that the return instruction in function B has two valid outgoing edges.

An attacker can manipulate this to inject a loop when function B is a dispatcher function. The attacker allows the first call to B to proceed normally on edge 1, returning on edge 2. The attacker sets up memory so that when B is called the second time, the return will follow edge 2 instead of the usual edge 4. That is, even though the code was originally intended as straight-line execution, there exists a back-edge that will be allowed by any static, stateless CFI policy without a shadow stack. A shadow stack would block the transfer along edge 2.

6.2 Turing-complete computation

CFI ensures that the execution flow of a program stays within a predefined CFG. CFI implicitly assumes that the attacker *must* divert from this CFG for successful exploitation. We demonstrate that an attacker can achieve Turing-complete computation while following the CFG. This is not directly one of the attacker goals outlined in Section 3, however it is often a useful step in achieving attacks [14].

Specifically, we show that a single call to `printf()` allows an attacker to perform Turing-complete computation, even when protected with a shadow stack. We dub this printf-oriented programming. In our evaluation, we found it was possible to mount this kind of attack against all but one binary (which rewrote their own limited version of printf).

Our attack technique is not specific to `printf()`: we have constructed a similar attack using `fputs()` which is widely applicable but requires a loop obtained in the control-flow graph (via loop injection or otherwise) to be

Turing-complete. See Appendix C.

6.2.1 Printf-oriented programming

When we control the arguments to `printf()`, it is possible to obtain Turing-complete computation. We show this formally in Appendix B by giving calls to `printf()` which create logic gates. In this section, we give the intuition behind our attacks by showing how an attacker can conditionally write a value at a given location.

Assume address C contains a condition value, which is an integer that is promised to be either zero or one. If the value is one, then we wish to store the constant X at target address T. That is, we wish to perform the computation `*T = *C ? X : *T`. We show how this can be achieved using one call to `printf()`.

To do this, the attacker supplies the specially-crafted format string "`%s%hhnQ%*d%n`" and passes arguments $(C, S, X-2, 0, T)$, defined as follows:

1. C — the address of the condition. While the "`%s`" format specifier expects a string, we pass a pointer to the condition value, which is either the integer 0 or the integer 1. Because of the little-endian nature of x86, the integer 1 contains the byte 0x01 in the first (low) byte and 0x00 in the second byte. This means that when we print it as a string, if the condition value is 1 then exactly one byte will be written out whereas if it is 0 then nothing will be be printed.
2. S — the address of the Q in the format string (i.e., the address of the format string, plus 6). The "`%hhn`" specifier will write a single byte of output consisting of the number of characters printed so far, and will write it on top of the Q in the format string. If we write a 0, the null byte, then the format string will stop executing. If we write a 1, the format string will keep going. It is this action which creates the conditional.
3. $X-2$ — the constant we wish to store, minus two. This specifies the number of bytes to pad in the integer which will be printed. It is the value we wish to save minus two, because two bytes will have already been printed.
4. 0 — an integer to print. We do not care that we are actually printing a 0, only the padding matters.
5. T — the target save location. At this point in time, we have written exactly X bytes to the output, so "`%n`" will write that value at the target address.

Observe that in this example, we have made use of a *self-modifying* format string.

6.2.2 Practical printf-oriented programming

The previous section assumed that the attacker has control of the format string argument, which is usually not the case. We show using simple techniques it is possible to achieve the same results without this control.

We first define the *destination* of a `printf()` call according to its type. The destination of an `sprintf()` call is the address the first argument points to (the destination buffer). The destination of a `fprintf()` call is the address of the temporary buffer in the `FILE` struct. The destination of a plain `printf()` call is the destination buffer of `fprintf()` when called with `stdout`.

Our attack requires three conditions to hold:

- the attacker controls the destination buffer;
- the format string passed to the call to `printf()` already contains a "`%s`" specifier; and,
- the attacker controls the argument to the format specifier as well as a few of the words further down on the stack.

We mount our attack by pointing the destination buffer on top of the stack. We use the "`%s`" plus the controlled argument to overwrite the pointer to the format string (which is stored on the stack), replacing it with a pointer to an attacker-controlled format string. We then skip past any uncontrolled words on the stack with harmless ``'%x'`` specifiers. We can then use the remaining controlled words to pivot the `va_list` pointer.

If we do not control any buffer on the stack, we can obtain partial control of the stack by continuing our arbitrary write with the `%s` specifier to add arguments to `printf()`. Note that this does not allow us to use null bytes in arguments, which in 64-bit systems in particular makes exploitation difficult.

6.3 Implications

Our analysis of fully-precise static CFI, the strongest imaginable static CFI policy, shows that preventing attackers with partial control over memory from gaining Turing-complete computation is almost impossible. Run-time libraries and applications contain powerful functions that are part of the valid CFG and can be used by attackers to implement their malicious logic. Attackers can use dispatcher functions to bend control flow within the valid CFG to reach these powerful functions.

Furthermore, we see that if an attacker can find one of these functions and control arguments to it, the attacker will be able to both write to and read from arbitrary addresses at multiple points in time. Defenses which allow attackers to control arguments to these functions must be able to protect against this stronger threat model.

7 Fully-Precise Static CFI Case Studies

We now look at some practical case studies to examine how well fully-precise static CFI can defend against real-

life exploits on vulnerable programs, both with and without a shadow stack. We split our evaluation into two parts. First, we show that attackers can indeed obtain arbitrary control over memory given actual vulnerabilities. Second, we show that given a program where the attacker controls memory at one point in time, it is possible to mount a control-flow bending attack. Our results are summarized in Table 2.

Our examples are all evaluated on a Debian 5 system running the binaries in x86 64-bit mode. We chose 64-bit mode because most modern systems are running as 64-bit, and attacks are more difficult on 64-bit due to the increased number of registers (data is loaded off of the stack less often).

We do not implement fully-precise static CFI. Instead, for each of our attacks, we manually verify that each indirect control-flow transfer is valid by checking that the edge taken occurs during normal program execution. Because of this, we do not need to handle dynamically linked libraries specially: we manually check those too.

7.1 Control over memory

The threat model we defined earlier allows the attacker to control memory at a single point in time. We argue that this level of control is achievable with most vulnerabilities, by analyzing four different binaries.

7.1.1 Nginx stack buffer overflow

We examined the vulnerability in CVE-2013-2028 [19]: a signedness bug in the chunked decoding component of nginx. We found it is possible to write arbitrary values to arbitrary locations, even when nginx is protected by fully-precise static CFI with a shadow stack, by modifying internal data structures to perform a control-flow bending attack.

The vulnerability occurs when an attacker supplies a large claimed buffer size, overflowing an integer and triggering a stack-based buffer overflow. An attacker can exploit this by redirecting control flow down a path that would never occur during normal execution. The Server Side Includes (SSI) module contains a call to `memcpy()` where all three arguments can be controlled by the attacker. We can arrange memory so after `memcpy()` completes, the process will not crash and will continue accepting requests. This allows us to send multiple requests and set memory to be exactly to the attacker's choosing.

Under benign usage, this `memcpy()` method is called during the parsing of a SSI file. The stack overflow allows us to control the stack and overwrite the pointer to the request state (which is passed on the stack) to point to a forged request structure, constructed to contain a partially-completed SSI structure. This lets us re-direct control flow to this `memcpy()` call. We are able to control its source and length arguments easily because they point to data on the heap which we control. The destination buffer is not typically under our control: it is obtained by the result of a call to nginx's memory allocator. However, we can cause the allocator to return a pointer to an arbitrary location by controlling the internal data structures of the memory allocator.

7.1.2 Apache off by one error

We examined an off-by-one vulnerability in Apache's handling of URL parameters [11]. We found that it is no longer exploitable in practice, when Apache is protected with CFI.

The specific error overwrites a single extra word on the stack; however, this word is not under the attacker's control. Instead, the word is a pointer to a string on the heap, and the string on the heap is under the attacker's control. This is a very contrived exploit, and it was not exploitable on the majority of systems in the first place due to the word on the stack not containing any meaningful data. However, on some systems the overwritten word contained a pointer to a data structure which contains function pointers. Later, one of these function pointers would be invoked, allowing for a ROP attack.

When Apache is protected with CFI, the attacker is not able to meaningfully modify the function pointers, and therefore cannot actually gain anything. CFI is effective in this instance because the attacker never obtains control of the machine in the first place.

7.1.3 Smbclient printf vulnerability

We examined a format string vulnerability in smbclient [26]. Since we already fully control the format string of a `printf()` statement, we can trivially control all of memory with printf-oriented programming.

7.1.4 Wireshark stack buffer overflow

A vulnerability in Wireshark's parsing of mpeg files allows an attacker to supply a large packet and overflow a stack buffer. We identify a method of creating a repeatable arbitrary write given this vulnerability even in the presence of a shadow stack.

The vulnerability occurs in the `packet_list_dissect_and_cache_record` function where a fixed-size buffer is created on the stack. An attacker can use an integer overflow to create a buffer of an arbitrary size larger than the allocated space. This allows for a stack buffer overflow.

We achieve an arbitrary write even in the presence of a shadow stack by identifying an arbitrary write in the `packet_list_change_record` function. Normally,

Binary	CFI without shadow stack				CFI with shadow stack			
	Arbitrary write	Info. leakage	Confined code execution	Arbitrary code execution	Arbitrary write	Info. leakage	Confined code execution	Arbitrary code execution
nginx	yes	write	dispatcher	dispatcher	yes	write	no	no
apache	no	write	printf	dispatcher	no	write	write	write
smbclient	yes	printf	printf	printf	yes	printf	printf	printf
wireshark	yes	printf	printf	dispatcher	yes	printf	write	write
xpdf	?	dispatcher	printf	dispatcher	?	write	printf	no
mysql	?	dispatcher	printf	dispatcher	?	write	printf	no

Table 2: The results of our evaluation of the 6 binaries. The 2nd and 6th columns indicate whether the vulnerability we examined allows an attacker to control memory. The other columns indicate which attack goals would be achievable, assuming the attacker controls memory. A "no" indicates that we were not able to achieve that attack goal; anything else indicates it is achievable, and indicates the attack technique we used to achieve the goal.

this would not be good enough, as this only writes a single memory location. However, an attacker can loop this write due to the fact that the GTK library method `gtk_tree_view_column_cell_set_cell_data`, which is on the call stack, already contains a loop that iterates an attacker-controllable number of times. These two taken together give full control over memory.

7.1.5 Xpdf & Mysql

For two of our six case studies, we were unable to reproduce the public exploit, and as such could not test if memory writes are possible from the vulnerability.

7.2 Exploitation assuming memory control

We now demonstrate that an attacker who can control memory at one point in time can achieve all three goals listed in Section 3, including the ability to issue attacker-desired system calls. (Our assumption is well-founded: in the prior section we showed this is possible.) Prior work has already shown that if arbitrary writes are possible (e.g., through a vulnerability) then data-only attacks are realistic [8]. We show that control-flow bending attacks that are not data-only attacks are also possible.

7.2.1 Evaluation of nginx

Assuming the attacker can perform arbitrary writes, we show that the attacker can read arbitrary files off of the server and relay them to the client, read arbitrary memory out of the server, and execute an arbitrary program with arbitrary arguments. The first two attack goals can be achieved even with a shadow stack; our third attack only works if there is no shadow stack. Nginx is the only binary which is not exploitable by printf-oriented programming, because nginx rewrote their own version of `printf()` and removed "`%n`".

An attacker can read any file that nginx has access to and cause their contents to be written to the output socket, using a purely non-control-data attack. For brevity, we do not describe this attack in detail: prior work has described that these types of exploits are possible.

Our second attack can be thought of as a more controlled version of the recent Heartbleed vulnerability [21], allowing the attacker to read from an *arbitrary* address and dump it to the attacker. The response handling in nginx has two main phases. First, it handles the header of the request and in the process initializes many structs. Then, it parses and handles the body of the request, using these structs. Since the vulnerability in nginx occurs during the parsing of the request body, we use our control over memory to create a forged struct that was not actually created during the initialization phase. In particular, we initialize the `postpone_filter` module data structure (which is not used under normal execution) with an internally-inconsistent state. This causes the module to read data from an arbitrary address of an arbitrary length and copy it to the response body.

Our final attack allows us to invoke `execve()` with arbitrary arguments, if fully-precise static CFI is used without a shadow stack. We use `memcpy()` as a dispatcher function to return into `ngx_sprintf()` and then again into `ngx_exec_new_binary()`, which later on calls `execve()`. By controlling its arguments, the attacker gets arbitrary code execution.

In contrast, when there is a shadow stack, we believe it is impossible for an attacker to trigger invocation of `execve()` due to privilege separation provided by fully-precise static CFI. The master process spawns children via `execve()`, but it is only ever called there — there is no code path that leads to `execve()` from any code point that is reachable within a child process. Thus, in this case CFI effectively provides a form of privilege separation for free, if used with a shadow stack.

7.2.2 Evaluation of apache

On Apache the attacker can invoke `execve()` with arbitrary arguments. Other attacks similar to those on nginx are possible; we omit them for brevity. When there is no shadow stack, we can run arbitrary code by using `strcat()` as a dispatcher gadget to return to a function which later invokes `execve()` under compilations which link the Windows `main` method. When there is a shadow stack, we found a loop that checks, for each module, if the module needs to be executed for the current request. By modifying the conditions on this loop we can cause `mod_cgi` to execute an arbitrary shell command under any compilation. Observe that this attack involves overwriting a function pointer, although to a valid target.

7.2.3 Evaluation of smbclient

Smbclient contains an interpreter that accepts commands from the user and sends them to a Samba fileserver. An attacker who controls memory can drive the interpreter to send any action she desired to the fileserver. This allows an attacker to perform any action on the Samba filesystem that the user could. This program is a demonstration that on some programs, CFI provides essentially no value due to the expressiveness of the original application.

This is one of the most difficult cases for CFI. The only value CFI adds to a binary is restricting it to its CFG: however, when the CFG is easy to traverse and gives powerful functions, CFI adds no more value than a system call filter.

7.2.4 Evaluation of wireshark

An attacker who controls memory can write to any file that the current user has access to. This gives power equivalent to arbitrary code execution by, for example, overwriting the `authorized_keys` file. This is possible because wireshark can save traces, and an attacker who controls memory can trivially overwrite the filename being written to with one the attacker picks.

If the attacker waits for the user to click save and simply overwrites the file argument, this would be a data-only attack under our definitions. It is also possible to use control-flow bending to invoke `file_save_as_cb()` directly, by returning into the GTK library and overwriting a code pointer with the file save method, which is within the CFG.

7.2.5 Evaluation of xpdf

Similar to wireshark, an attacker can use xpdf to write to arbitrary files using `memcpy()` as a dispatcher gadget when there is no shadow stack. When a shadow stack is present, we are limited to a printf-oriented programming attack and we can only write files with specific extensions, which does not obviously give us ability to run arbitrary code.

7.2.6 Evaluation of mysql

When no shadow stack is present, attacks are trivial. A dispatcher gadget lets us return into `do_system()`, `do_exec()`, or `do_perl()` from within the mysql client. (For this attack we assume a vulnerable client to connects to a malicious server controlled by the attacker.) When a shadow stack is present the attacker is more limited, but we still can use printf-oriented programming to obtain arbitrary computation on memory. We could not obtain arbitrary execution with a shadow stack.

7.3 Combining attacks

As these six case studies indicate, control-flow bending is a realistic attack technique. In the five cases where CFI does not immediately stop the exploit from occurring, as it does for Apache, an attacker can use the vulnerability to achieve arbitrary writes in memory. From here, it is possible to mount traditional data-only attacks (e.g., by modifying configuration data-structures). We showed that using control-flow bending techniques, more powerful attacks are possible. We believe this attack technique is general and can be applied to other applications and vulnerabilities.

8 Related work

Control-flow integrity. Control-flow integrity was originally proposed by Abadi et al. [1, 15] a decade ago. Classical CFI instruments indirect branch target locations with equivalence-class numbers (encoded as a label in a side-effect free instruction) that are checked at branch locations before taking the branch. Many other CFI schemes have been proposed since then.

The most coarse-grained policies (e.g., Native Client [40] or PittSFIeld [20]) align valid targets to the beginning of chunks. At branches, these CFI schemes ensure that control-flow is not transferred to unaligned addresses. Fine-grained approaches use static analysis of source code to construct more accurate CFGs (e.g., WIT [2] and HyperSafe [39]). Recent work by Niu et al. [27] added support for separate compilation and dynamic loading. Binary-only CFI implementations are generally more coarse-grained: MoCFI [13] and BinCFI [44] use static binary rewriting to instrument indirect branches with additional CFI checks.

CFI evaluation metrics. Others have attempted to create methods to evaluate practical CFI implementations. The Average Indirect target Reduction (AIR) [44] metric

was proposed to measure how much on average the set of indirect valid targets is reduced for a program under CFI. We argue that this metric has limited utility, as even high AIR values of 99% are insecure, allowing an attacker to perform arbitrary computation and issue arbitrary system calls. The gadget reduction metric is another way to evaluate CFI effectiveness [27], by measuring how much the set of reachable gadgets is reduced overall. Gadget finder tools like ROPgadget [33] or ropper [34] can be used to estimate this metric.

CFI security evaluations. There has recently been a significant effort to analyze the security of specific CFI schemes, both static and dynamic. Göktaş et al. [16] analyzed the security of static coarse-grained CFI schemes and found that the specific policy of requiring returns to target call-preceded locations is insufficient. Following this work, prevent-the-exploit-style coarse-grained CFI schemes with dynamic components that rely on runtime heuristics were defeated [5, 14]. The attacks relied upon the fact that the attacks could hide themselves from the dynamic heuristics, and then reduced down to attacks on coarse-grained CFI. Our evaluation of minimal programs builds on these results by showing that coarse-grained CFI schemes which have an AIR value of 99% are still vulnerable to attacks on trivially small programs.

Non-control data attacks. Attacks that target only sensitive data structures were categorized as *pure data attacks* by Pincus and Baker [32]. Typically, these attacks would overwrite application-specific sensitive variables (such as the "is authenticated" boolean which exists within many applications). This was expanded by Chen et al. [8] who demonstrated that *non-control data attacks* are practical attacks on real programs. Our work generalizes these attacks to allow modifications of control-flow data, but only in a way that follows the CFI policy.

Data-flow integrity. Nearly as old of an idea as CFI, Data-Flow Integrity (DFI) provides guarantees for the integrity of the data within a program [6]. Although the original scheme used static analysis to compute an approximate data-flow graph — what we would now call a coarse-grained approach — more refined DFI may be able to protect against our attacks. We believe security evaluation of prevent-the-corruption style defenses such as DFI is an important future direction of research.

Type- and memory-safety. Other defenses have tried to bring type-safety and memory-safety to unsafe languages like C and C++. SoftBound [22] is a compile-time defense which enforces spatial safety in C, but at a 67% performance overhead. CETS [23] extends this work with a compile-time defense that enforces temporal safety in C, by protecting against memory management errors. CCured [24] adds type-safe guarantees to C by attempting to statically determine when errors cannot occur, and dynamically adding checks when nothing can be proven statically. Cyclone [17] takes a more radical approach and re-designs C to be type- and memory-safe. Code-Pointer Integrity (CPI) [18] reduces the overhead of SoftBound by only protecting code pointers. While CPI protects the integrity of all indirect control-flow transfers, limited control-flow bending attacks using conditional jumps may be possible by using non-control-data attacks. Evaluating control-flow bending attacks on CPI would be an interesting direction for future work.

9 Conclusion

Control-flow integrity has historically been considered a strong defense against control-flow hijacking attacks and ROP attacks, if implemented to its fullest extent. Our results indicate that this is not entirely the case, and that control-flow bending allows attackers to perform meaningful attacks even against systems protected by fully-precise static CFI. When no shadow stack is in place, dispatcher functions allow powerful attacks. Consequently, CFI without return instruction integrity is not secure. However, CFI with a shadow stack does still provide value as a defense, if implemented correctly. It can significantly raise the bar for writing exploits by forcing attackers to tailor their attacks to a particular application; it limits an attacker to issue only system calls available to the application; and it can make specific vulnerabilities unexploitable under some circumstances.

Our work has several implications for design and deployment of CFI schemes. First, shadow stacks appear to be essential for the security of CFI. We also call for adversarial analysis of new CFI schemes before they are deployed, as our work indicates that many published CFI schemes have significant security weaknesses. Finally, to make control-flow bending attacks harder, deployed systems that use CFI should consider combining CFI with other defenses, such as data integrity protection to ensure that data passed to powerful functions cannot be corrupted in the presence of a memory safety violation.

More broadly, our work raises the question: just how much security can prevent-the-exploit defenses (which allow the vulnerability to be triggered and then try to prevent exploitation) provide? In the case of CFI, we argue the answer to this question is that it still provides some, but not complete, security. Evaluating other prevent-the-exploit schemes is an important area of future research.

We hope that the analyses in this paper help establish a basis for better CFI security evaluations and defenses.

10 Acknowledgments

We would like to thank Jay Patel and Michael Theodorides for assisting us with three of the case studies. We

would also like to thank Scott A. Carr, Per Larsen, and the anonymous reviewers for countless discussions, feedback, and suggestions on improving the paper. This work was supported by NSF grant CNS-1513783, by the AFOSR under MURI award FA9550-12-1-0040, and by Intel through the ISTC for Secure Computing.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *CCS'05* (2005).

[2] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *IEEE S&P'08* (2008).

[3] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *ACSAC'11* (2011).

[4] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11* (2011).

[5] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security'14* (2014).

[6] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *OSDI '06* (2006).

[7] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *CCS'10* (2010), pp. 559–572.

[8] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security'05* (2005).

[9] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *NDSS'14* (2014).

[10] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security'98* (1998).

[11] COX, M. CVE-2006-3747: Apache web server off-by-one buffer overflow vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747`, 2006.

[12] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE S&P'14* (2014).

[13] DAVI, L., DMITRIENKO, R., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NUERNBERGER, S., AND SADEGHI, A. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS'12* (2012).

[14] DAVI, L., SADEGHI, A.-R., LEHMANN, D., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security'14* (2014).

[15] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *OSDI'06* (2006).

[16] GOKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *IEEE S&P'14* (2014).

[17] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *ATC'02* (2002).

[18] KUZNETSOV, V., PAYER, M., SZEKERES, L., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI'14* (2014).

[19] MACMANUS, G. CVE-2013-2028: Nginx http server chunked encoding buffer overflow. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028`, 2013.

[20] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *USENIX Security'06* (2006).

[21] MEHTA, N., RIKU, ANTTI, AND MATTI. The Heartbleed bug. `http://heartbleed.com/`, 2014.

[22] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI'09* (2009).

[23] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: Compiler enforced temporal safety for C. In *ISMM'10* (2010).

[24] NECULA, G., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*, 3 (2005), 477–526.

[25] NERGAL. The advanced return-into-lib(c) exploits. *Phrack 11*, 58 (Nov. 2007), `http://phrack.com/issues.html?issue=67&id=8`.

[26] NISSL, R. CVE-2009-1886: Formatstring vulnerability in smbclient. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886`, 2009.

[27] NIU, B., AND TAN, G. Modular control-flow integrity. In *PLDI'14* (2014).

[28] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security* (2013), pp. 447–462.

[29] PAX-TEAM. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[30] PAYER, M., BARRESI, A., AND GROSS, T. R. Fine-grained control-flow integrity through binary hardening. In *DIMVA'15*.

[31] PHILIPPAERTS, P., YOUNAN, Y., MUYLLE, S., PIESSENS, F., LACHMUND, S., AND WALTER, T. Code pointer masking: Hardening applications against code injection attacks. In *DIMVA'11* (2011).

[32] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2* (2004), 20–27.

[33] SALWAN, J. ROPgadget – Gadgets finder and auto-roper. `http://shell-storm.org/project/ROPgadget/`, 2011.

[34] SCHIRRA, S. Ropper – rop gadget finder and binary information tool. `https://scoding.de/ropper/`, 2014.

[35] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *USENIX Security'11* (2011).

[36] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07*.

[37] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *IEEE S&P'13* (2013).

[38] VAN DE VEN, A., AND MOLNAR, I. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[39] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P'10* (2010).

[40] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P'09* (2009).

[41] ZENG, B., TAN, G., AND ERLINGSSON, U. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security'13* (2013).

[42] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., MCCAMANT, S., AND SZEKERES, L. Protecting function pointers in binary. In *ASIACCS'13* (2013).

[43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE S&P'13* (2013).

[44] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *USENIX Security'13* (2013).

A Minimal vulnerable program for indirect jump or call hijacking

The program in Figure 6 contains a bug that allows the attacker to reliably hijack an indirect jump or indirect call target. The function `overflow()` allows an attacker to overflow a struct allocated on the stack that contains two pointers used as the targets for an indirect jump or an indirect call, respectively. The attacker can use the indirect jump or call to divert control flow to a return gadget and continue with a classic ROP attack. Alternatively, an attacker may rely on JOP or COP techniques. We also examined variations of this minimal vulnerable program, e.g., putting the struct somewhere on the heap or requiring the attacker to first perform a stack pivot to ensure that the stack pointer points to attacker-controlled data.

```
#include <stdio.h>
#include <string.h>
#define STDIN 0

void jmptarget();
void calltarget();

struct data {
 char buf[1024];
 int arg1;
 int arg2;
 int arg3;
 void (*jmpPtr)();
 void (*callPtr)(int,int,int);
};

void overflow() {
 struct data our_data;
 our_data.jmpPtr = &&label;
 our_data.callPtr = &calltarget;
 printf("%x\n", (unsigned int)&our_data.buf);
 printf("\ndata> ");
 read(STDIN, our_data.buf, 1044);
 printf("\n");
 asm("push %0; push %1; push %2; call *%3; add $12,%%esp;"
 : : "r"(our_data.arg3),
     "r"(our_data.arg2),
     "r"(our_data.arg1),
     "r"(our_data.callPtr));
 asm("jmp *%0" : : "r"(our_data.jmpPtr));
 printf("?\n");
label:
 printf("label reached\n");
}

void jmptarget() {
 printf("jmptarget() called\n");
}

void calltarget(int arg1, int arg2, int arg3) {
 printf("calltarget() called (args: %x, %x, %x)\n",
 arg1, arg2, arg3);
}

int main(int argc, char* argv[]) {
 setbuf(stdout, NULL);
 overflow();
 printf("\ndone.\n");
 return 0;
}
```

Figure 6: A minimal vulnerable program that allows hijack of an indirect jump or indirect call target.

B Printf is Turing-complete

The semantics of `printf()` allow for Turing-complete computation while following the minimal CFG.

At a high level, we achieve Turing-completeness by creating logic gates out of calls to `printf()`. We show how to expand a byte to its eight bits, and how to compact the eight bits back to a byte. We will compute on values by using them in their base-1 (unary) form and we will use string concatenation as our primary method of arithmetic. That is, we represent a true value as the byte sequence 0x01 0x00, and the false value by the byte sequence 0x00 0x00, so that when treated as strings their lengths are 1 and 0 respectively.

Figure 7 contains an implementation of an OR gate using only calls to `printf()`. In the first call to `printf()`, if either of the two inputs is non-zero, the output length will be non-zero, so the output will be set to a non-zero value. The second call to `printf()` normalizes the value so if it was any non-zero value it becomes a one. Figure 7 implements a NOT gate using the fact that adding 255 is the same as subtracting one, modulo 256.

In order to operate on bytes instead of bits in our contrived format, we implement a test gate which can test if a byte is equal to a specific value. By repeating this test gate for each of the 256 potential values, we can convert a 8-bit value to its "one-hot encoding" (a 256-bit value with a single bit set, corresponding to the original value). Splitting a byte into bits does not use a pointer to a byte, but a byte itself. This requires that the byte is on the stack. Moving it there takes some effort, but can still be done with `printf()`. The easiest way to achieve this would be to interweave calls to `memcpy()` and `printf()`, copying the bytes to the stack with `memcpy()` and then operating on them with

```
void or(int* in1, int* in2, int* out) {
  printf("%s%s%n", in1, in2, out);
  printf("%s%n", out, out);
}

void not(int* in, int* out) {
  printf("%*d%s%n", 255, in, out);
  printf("%s%n", out, out);
}

void test(int in, int const, int* out) {
  printf("%*d%*d%n", in, 0, 256-const, 0, out);
  printf("%s%n", out, out);
  printf("%*d%s%n", 255, out, out);
  printf("%s%n", out, out);
}

char* pad = memalign(257, 256);
memset(pad, 1, 256);
pad[256] = 0;
void single_not(int* in, int* out) {
  printf("%*d%s%n%hhn%s%s%n", 255, in, out,
         addr_of_argument, pad, out, out);
}
```

Figure 7: Gadgets for logic gates using printf.

`printf()`. However, whis requires more of the program CFG, so we instead developed a technique to achieve the same goal without resorting to `memcpy()`. When `printf()` is invoked, the characters are not sent directly to the stdout stream. Instead, `printf()` will use the `FILE` struct corresponding to the stdout stream to buffer the data temporarily. Since the struct is stored in a writable memory location, the attacker can invoke `printf()` with the "%n" format specifier to point the buffer onto the stack. Then, by reading values out of memory with "%s" the attacker can move these values onto the stack. Finally, the buffer can be moved back to its original location.

It is possible to condense multiple calls to `printf()` to only one. Simply concatenating the format strings is not enough, because the length of the strings is important with the "%n" modifier. That is, after executing a NOT gate, the string length will either be 255 or 256. We cannot simply insert another NOT gate, as that would make the length be one of 510, 511, or 512. We fix this by inserting a *length-repairing sequence* of "%hhn%s", which pads the length of the string to zero modulo 256. We use it to create a NOT gate in a single call to `printf()` in Figure 7. Using this technique, we can condense an arbitrary number of gates into a single call to `printf()`. This allows bounded Turing-complete computation.

To achieve full Turing-complete computation, we need a way to loop a format string. This is possible by overwriting the pointer inside `printf()` that tracks which character in the format string is currently being executed. The attacker is unlucky in that at the time the "%n" format specifier is used, this value is saved in a register on our 64-bit system. However, we identify one point in time in which the attacker can always mount the attack. The `printf()` function makes calls to `puts()` for the static components of the string. When this function call is made, all registers are saved to the stack. It turns out that an attacker can overwrite this pointer from within the `puts()` function. By doing this, the format string can be looped.

An attacker can cause `puts()` to overwrite the desired pointer. Prior to `printf()` calling `puts()`, the attacker uses "%n" format specifiers to overwrite the stdout `FILE` object so that the temporary buffer is placed directly on top of the stack where the index pointer will be saved. Then, we print the eight bytes corresponding to the new value we want the pointer to have. Finally, we use more "%n" format specifiers to move the buffer back to some other location so that more unintended data will not be overwritten.

C Fputs-oriented programming

These `printf`-style attacks are not unique to `printf()`: many other functions can be exploited in a similar manner. We give one further attack using `fputs()`. For brevity, we show how an attacker can achieve a conditional write, however other computation is possible.

The `FILE` struct contains three `char*` fields to temporarily buffer character data before it is written out: a base pointer, a current pointer, and an end pointer. `fputs()` works by storing bytes sequentially starting from the base pointer keeping track with the current pointer. When it exceeds the end pointer, the data is written out, and the current pointer is set back to the base. Programmatically, the way this works is that if the current pointer is larger than the end pointer, `fputs()` flushes the buffer and then sets the current pointer to the base pointer and continues writing.

This can be used to conditionally copy from source address S to target address T if the byte address C is nonzero. Using `fputs()`, the attacker copies the byte at C on top of each of the 8 bytes in the end pointer. Then, the attacker sets the current pointer to T and then calls `fputs()` with this `FILE` and argument S. If the byte at C is zero, the end pointer is the NULL pointer, and no data is written. Otherwise, the data is written.

This attack requires two calls to `fputs()`. We initialize memory with the constant pointers that are desired. The first call to `fputs()` moves the C byte over the end pointer. The second call is the conditional move. The two calls can be obtained by loop injection, or by identifying an actual loop in the CFG.