Evaluating Control-Flow Integrity with Syscall Reachability Analysis

by

Tony Liao

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2022 by Tony Liao

Evaluating Control-Flow Integrity with Syscall Reachability Analysis

Tony Liao
Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto
2022

Abstract

In order to defend against memory exploits, control-flow integrity (CFI) has been widely researched as a defence and seen significant industry adoption. While attacks have demonstrated its limitations in preventing control-flow hijacking, we argue that CFI can still prevent end-to-end exploits by limiting an attacker's access to the syscall interface. We design and implement a specialization of static taint analysis called *syscall reachability analysis* to over-approximate the set of *syscall gadgets* available to the attacker under different CFI policies, which we propose as a quantitative upper bound on *exploitability*. Evaluating over a set of representative C/C++ programs, We find that most programs do not make very many *sensitive* syscalls and that in most of the remaining cases examined, the harm can be mitigated through sanitization of syscall arguments.

Acknowledgements

I would like to thank my advisor, Professor David Lie, for helping me navigate through the fog. His patience, experience, and consistency was the pillar of this project. I've learned a great deal from him over the last two years — not just about security but about questioning assumptions, asking the right questions, and facing problems with integrity.

I'd also like to thank my lab-mates Wei, John, Eric, and Yuqin for their guidance, thoughtful discussion, and sometimes obscure interests. Despite starting my degree during a pandemic, we had some good times together, and I really am going to miss everyone.

I'd like to thank my parents for always being there for me, and never questioning that I really was busy with school (even if I sometimes wasn't).

Lastly, I am grateful to the University of Toronto, the Government of Ontario, and Huawei Technologies for the financial support that enabled this project.

Contents

1 Introduction 1

2 Background and Related Work 4

2.1 Return-Oriented Programming 4
2.2 Control-Flow Integrity 5
2.3 Sensitive Syscalls 6
2.4 Taint Analysis 6

3 Design 7

3.1 Analysis Objective 7
3.2 Threat Model 9
3.3 CFI Model 10
3.4 Syscall Reachability 10
3.4.1 Modular Analysis for Shared Libraries 12
3.5 Sources of Imprecision 12

4 Implementation 14

4.1 Workflow 14
4.2 Control-Flow Graph Pruning 15

5 Evaluation 17

5.1 Experiment Setup 17
5.2 Summary Statistics 18
5.2.1 Indirect Call Site Analysis 21
5.3 Syscall Usage 24
5.4 Case Studies 26
5.4.1 NGINX 26
5.4.2 Wireshark 28
5.4.3 OpenVPN3 29
5.5 Analysis Cost 30

6 Limitations and Future Work 32

7 Conclusion 33

Bibliography 34

List of Tables

5.1 Test program suite . . . 18
5.2 Summary statistics for binaries . . . 20
5.3 Summary statistics for client-library analysis . . . 22
5.4 Distribution of exploitable sinks . . . 23
5.5 Analysis time and peak memory usage . . . 31

List of Figures

1.1 Triggering a syscall under CFI 2

3.1 Steps in an abstracted end-to-end memory exploit 8

List of Listings

3.1 A toy program illustrating attacker capabilities 9
5.1 Source code for an exploitable exec in `nginx` 26
5.2 Source code for a false positive mmap in `nginx`. 27
5.3 Source code for an unexploitable exec in `wireshark` 28
5.4 Source code for a false positive write in `openvpn3` 29

Chapter 1

Introduction

Memory corruption vulnerabilities pose one of the largest threats to software today. After decades of research in attacks and defenses, these bugs remain common and dangerous [1], leading to remote execution in the worst case. They arise because weakly typed languages like C and C++ allow access to memory that is out of bounds or no longer valid, causing *undefined behaviour*. Although type-/memory-safe languages exist, C and C++ continue to be used when high performance, efficient use of memory, access to low-level interfaces, and compatibility with legacy code are desired. While hardware acceleration promises to bring down the performance overhead of type/memory safety [31], today it is still prohibitively expensive for demanding applications.

The earliest memory exploits used simple code injection: The attacker places the code they want to run, typically a short instruction sequence to open a shell (called a *shellcode*), inside a buffer. They then trigger the memory vulnerability to overwrite a code pointer (e.g. a return address) with the address of the shellcode; the shellcode is run when this code pointer is dereferenced (e.g. on a return). The introduction of non-executable pages (NX) prevented this by requiring that writable pages not also be executable. This successfully separates code and data, but still allows *code reuse* attacks where existing code (particularly libc) is repurposed by the attacker to carry out an exploit. Modern code reuse attacks use a technique called *return-oriented programming* (ROP) [23], where typically short sequences of instructions followed by a return (called *ROP gadgets*) are chained together to perform the task of the shellcode. To construct this chain, the attacker overwrites a contiguous set of return addresses on the stack with the address of each gadget. A variant of ROP called *jump-oriented programming* (JOP) [7] uses jump/call instructions to chain together gadgets instead of returns, allowing it to bypass defences based on checking or protecting return addresses.

In order to prevent ROP/JOP and control-flow hijacking in general, control-flow integrity (CFI) [2] has been widely researched as a defence. It involves using static analysis to over-approximate a program's control-flow graph (CFG), and checking at each indirect control-flow transfer that the corresponding edge exists in the CFG. For instance, a common CFI policy requires that all indirect calls target the start of a function, which eliminates JOP gadgets that begin in the middle of a function. The tightness of this over-approximation is referred to as the *precision* or *granularity* of the CFI policy: The higher the precision, the fewer gadgets there are. The hope is that with sufficiently precise CFI, there will be so few gadgets as to make exploitation impossible, or at least infeasibly difficult. CFI has seen significant industry adoption in recent years: It has been integrated

into LLVM [9], WebAssembly [22], and the Android Linux kernel [10]. A newer trend has been the use of custom hardware to reduce the performance overhead of CFI enforcement, as in Intel CET [24] and ARM PA [19].

Despite this interest, the degree of protection offered by CFI is not clear. While fully-precise CFI for return addresses in the form of a shadow call stack [2] can prevent ROP entirely, JOP-style exploits are still possible: Attacks circumventing CFI have been demonstrated based on stitching together function-sized gadgets [12], abusing virtual function tables used for dynamic dispatch [21], permissive functions like `printf` and limitations of static analysis [6], and the imprecision of *scalable* pointer analysis [13]. Beyond precision metrics like equivalence class size and AIR [5], the effectiveness of CFI at preventing an end-to-end exploit is not well understood.

To address this, we turn to the syscall interface: In every remote userspace memory exploit, the last step is to trigger a syscall to compromise the system in some way (privilege escalation, data exfiltration, etc.) For code injection, this is done with a syscall instruction (`syscall`/`int 0x80` on x86 Linux) in the shellcode; for ROP, this is done by jumping to a syscall instruction somewhere in existing code. However, we observe that CFI can make triggering a syscall significantly harder: If every indirect call must target the start of a function, an attacker will generally not be able to choose the syscall number because it will typically be set to a constant in the function (e.g. calling `execve()` on x86 will have the syscall number hard-coded to 59). Instead, they will have to find a *syscall gadget* to make their syscall of choice. In libc, common syscalls are made available through wrapper functions (`execve()`, `execl()`, `open()`, `fopen()`, etc.), which can serve as syscall gadgets. However, if we add the modest assumption that no indirect call can target a syscall wrapper, this rule can be enforced by CFI (assuming GOT/PLT protection [15]). An attacker would therefore have to find a syscall wrapper call site in the binary to act as a syscall gadget. This model is shown in Figure 1.1.

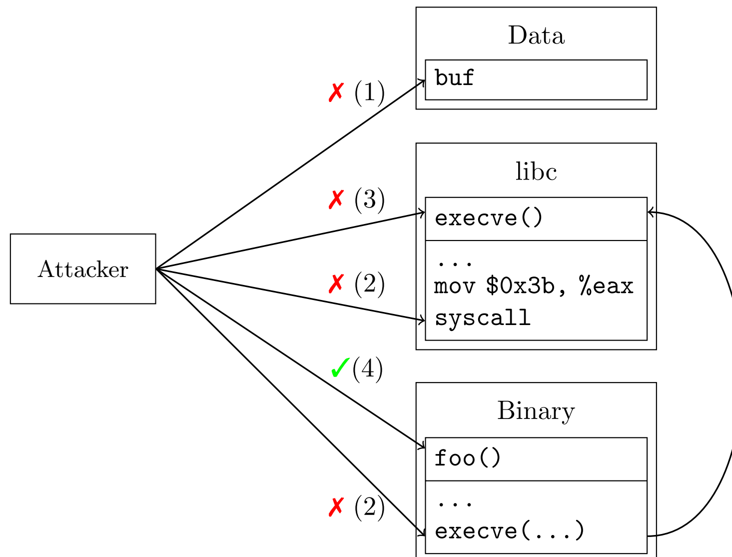

Figure 1.1: Triggering a syscall under CFI. (1) Execution of data pages is blocked by NX; (2) calling into the middle of a function is blocked by function-granularity CFI; (3) we additionally assume that libc syscall wrappers are not legal indirect call targets; (4) a syscall gadget must be a function in the binary that eventually calls into a libc syscall wrapper.

Under this model, *we propose to use the prevalence of syscall gadgets to measure the exploitability of CFI-protected programs.* Unlike previous efforts to evalulate CFI which examine an attacker's ability to string together gadgets in order to perform computation, we examine the effectiveness of the process sandbox at containing the attacker. By defining exploitability at the syscall interface, we restrict ourselves to a small and well-defined attack surface. Instead of considering all possible memory vulnerabilities or all possible ROP gadgets, we only consider a handful of *sensitive syscalls* capable of damaging the system (`execve`, `mmap`, `open`, `write`, and variants). In accordance with the end-to-end principle, evaluating the last stage in the exploitation pipeline allows us to fold all prior stages into the threat model: We simply assume the attacker has performed arbitrary computation in userspace, but has yet to issue a syscall and must follow the CFI policy.

With this attacker model, we use a specialization of static taint analysis to find syscall gadgets which we call *syscall reachability analysis*. In this analysis, indirect call targets allowed by the CFI policy are taint sources and sensitive syscall call sites are taint sinks; we use taint to represent dataflow that may be controlled by the attacker. If a syscall call site has a tainted argument, we call that call site *exploitable* and the corresponding code path a syscall gadget. By considering a variety of CFI policies, we evaluate the impact that CFI precision has on exploitability, using the number of exploitable syscalls as a quantitative upper bound.

We implement a prototype of this analysis with the LLVM toolchain, using SVF [28] to create a whole-program data-flow graph. We then apply the analysis to a suite of representative programs written in C and C++, evaluating program exploitability, analysis precision, and implementation scalability. In doing so, we see a clear difference in exploitability between CFI policies, and significant variation due to program function and structure. In most programs, many syscalls are unexploitable; for the rest, we argue that their harm can be mitigated through simple input sanitization in most of the cases examined. The analysis provides rich information about exploitable data-flow paths, and allows such a defence to be tailored to the program as needed.

In summary, we make the following contributions:

- The notion of exploitability under different CFI policies based on the prevalence of *syscall gadgets*;
- a specialization of static taint analysis for finding syscall gadgets under these CFI policies, which we call *syscall reachability analysis*;
- a prototype implementation of the analysis using SVF and LLVM, with optimizations for scalability;
- an evaluation of exploitability, precision, and scalability over a suite of representative programs.

Chapter 2

Background and Related Work

In this chapter, we provide background on an exploit technique called return-oriented programming (§2.1), and a defence called control-flow integrity which aims to prevent it (§2.2). We then discuss related work around the idea of a sensitive syscall (§2.3) and give a brief introduction to taint analysis (§2.4).

2.1 Return-Oriented Programming

Early exploits of memory vulnerabilities worked through direct code injection: The attacker crafts a short code sequence to open a shell (called a *shellcode*), places this code into a buffer, and triggers the memory vulnerability to jump to the shellcode. Modern systems have a feature called *non-executable pages* (NX) which marks writable pages as non-executable, preventing the jump into the buffer. While this eliminates the possibility of code injection, it does not prevent *code reuse*: The attacker can still use portions of the existing code to perform the task of the shellcode.

Return-oriented programming (ROP) [23] is a code reuse attack which uses sequences of instructions that end in a return, called *ROP gadgets*, to carry out desired operations. For instance, the x86 sequence [`mov $0x0, %rax; ret`] (corresponding to `return 0`) has the effect of setting `rax` to 0; [`add $0x1, %rdx; mov %rdx, %rax; ret`] adds 1 to `rdx` and copies it into `rax`. Sequences can also begin on mis-aligned instructions if the instruction set is variable-length. The attacker finds these sequences in existing code (typically libc), combines them to form a *ROP chain*, and triggers this chain by overwriting a contiguous sequence of return addresses on the stack with the address of each gadget.

Challenges in doing this include (1) finding a sufficient set of useful gadgets, and (2) figuring out how to combine them to perform the desired exploit (e.g. spawning a shell or writing to a sensitive file). The former is typically done by searching common libraries like libc using a set of heuristics; the latter depends on the type of exploit. With enough gadgets, it is possible to perform arbitrary userspace computation, allowing any exploit by creating a compiler to target this ROP instruction set. In order to trigger the ROP chain, the attacker also has to find the location of the gadgets in virtual memory, which is randomized by a widely deployed defence called address space layout randomization (ASLR); this can by done by leaking the library's random offset through another memory vulnerability or sometimes brute-force guessing. The last step in the exploit is to trigger

a syscall by jumping to either a syscall instruction or a function which issues a syscall (e.g. libc `execve()`). Today, ROP is mature: Tools exist for automating gadget discovery and ROP chain construction [20].

A variant of ROP called *jump-oriented programming* (JOP) [7] uses jumps and calls to string together gadgets instead of returns. This can be done on x86 using gadgets that end in a `pop-jmp` pair, which acts like a return. The benefit of JOP is that it bypasses defences based on protecting or monitoring return instructions.

2.2 Control-Flow Integrity

Control-flow integrity (CFI) [2] is a software defence which limits an attacker's ability to hijack control flow. It uses static analysis to over-approximate a program's control-flow graph (CFG), and checks at each code pointer (pointer into the code region) dereference that the corresponding edge exists in the CFG (direct calls are not checked because page permissions guarantee code integrity). The set of rules enforced by CFI is called the *CFI policy*. For example, a simple policy could require that all calls target the start of a function and that all returns target a call site. This reduces the set of ROP/JOP gadgets by eliminating the ones that call into the middle of a function or return to the middle of a basic block. However, this is clearly an over-approximation: At an indirect call site, in addition to the callee being a valid function, its function signature should match the caller's; as a lower bound, there exists some minimal set of legal targets that are possible in benign execution. The tightness of this over-approximation is referred to as the *precision* or *granularity* of the CFI policy.

CFI divides code pointers into two categories: *Forward edges* refer to C-style function pointer dereferences, C++ dynamic dispatch, and indirect jumps; *backward edges* refer to function returns. Since return addresses are immutable under benign execution, the backward-edge problem can be fully solved by isolating them (safe stack [18]) or creating an isolated copy (shadow stack [2]). This completely prevents ROP by guaranteeing the last-in first-out semantics of the call stack. However, achieving high precision in forward-edge CFI to prevent JOP-style exploits remains an open problem. Attacks against CFI have been demonstrated based on stitching together function-sized gadgets [12], abusing virtual function tables used for dynamic dispatch [21], permissive functions like `printf` and limitations of static analysis [6], and the imprecision of *scalable* pointer analysis [13]. In response to these attacks, recent CFI defences have proposed *context-sensitive* or *stateful* policies to improve precision beyond what is possible for a purely static policy [16, 17]; we consider these proposals orthogonal to our work.

While these attacks demonstrate the limitations of CFI in preventing control-flow hijacking, they often rely on specific code patterns or vulnerabilities to perform an end-to-end attack. For instance, the exploits in the Control-Flow Bending paper [6] use a legal indirect target that `exec`'s an arbitrary string on Apache httpd, and require that Wireshark be able to write to arbitrary files (including `.ssh/authorized_keys`). On Linux, Counterfeit OOP [21] overwrites a function pointer with the address of `system()`, which could be prevented by disallowing indirect calls to syscall wrappers. To futher examine this, we propose a static analysis to find the syscall gadgets that enable the critical last step of an exploit. We believe we are the first to evaluate CFI security through the exploitability of syscalls. Our analysis is similar to Control Jujutsu's [13] in that both use dataflow analysis to find

gadgets under an explicit CFI model. They differ in that instead of under-approximating the set of ACICS gadgets in order to construct an exploit, ours over-approximates the set of syscall gadgets in order to evaluate exploitability.

2.3 Sensitive Syscalls

The idea of a *security-sensitive syscall* has been explored by defences which place a reference monitor around the syscall interface. Such a monitor can be implemented by instrumenting the binary, modifying the kernel syscall handler, or interposing on all syscalls with a mechanism like eBPF [4]. ROPecker [8] and Intel-PT-based CFI defences like GRIFFIN [14] check for violations of their security policies asynchronously, and block on sensitive syscalls until this check has been completed. The set of sensitive syscalls depends on the attacks under consideration: The typical concern is privilege escalation through spawning a shell or changing page permissions to allow code injection. We use a similar set of syscalls in our analysis in order to evaluate exploitability through these mechanisms.

2.4 Taint Analysis

Taint analysis is a type of data-flow analysis that tracks information flows from *sources* to *sinks*. A variable is considered to be tainted by a source if it is derived from that source or, transitively, another variable tainted by that source. The meaning of taint depends on the use case: It could indicate an untrusted value like unsanitized user input, or secret information like a value derived from a password. Since taint analysis reasons about the provenance of memory objects rather than their values, it tends to be lightweight compared to more general techniques like symbolic execution. It has been used in a static context for finding bugs and malicious code, particularly on Android [3]; runtime defenses have also been proposed based on dynamic taint *tracking* [11, 27].

Chapter 3

Design

In this chapter, we describe challenges in exploiting memory bugs under CFI and state our goal: To find an upper bound on exploitability based on *syscall gadgets*, assuming a strong attacker (§3.1). We then give more formal models of attack (§3.2) and defence (§3.3), and propose a static taint analysis to find syscall gadgets which we call *syscall reachability analysis* (§3.4). Lastly, we describe simplifying assumptions needed to make the analysis work, and possible errors that may result from them (§3.5).

3.1 Analysis Objective

An end-to-end exploit of a memory vulnerability in a userspace process can generally be divided into three distinct phases: (1) Triggering the vulnerability to corrupt control-relevant data, (2) stringing together gadgets to set up syscall arguments, and (3) triggering a syscall to compromise the system. While many types of memory vulnerabilities exist (buffer overflow, use-after-free, etc.), we can represent them abstractly with an attacker who has an *arbitrary write primitive*. Such an attacker can set writable memory to an arbitrary state at some point in time, but can not directly change data registers or the program counter. Similarly, the ability to form a gadget chain can be abstracted with an attacker who can perform Turing-complete computation. By "Turing-complete" we mean that the attacker can set both writable memory and data registers to an arbitrary state at some point in time, but must still respect the separation between user and kernel modes (unprivileged ISA, page permissions). These abstracted phases are shown in Figure 3.1.

While performing step 2 under CFI has been explored in the literature (§2.2), step 3 has received comparatively little attention. In ROP/JOP (§2.1), step 3 is straightforward: Having already gained arbitrary computation capabilities, the attacker prepares arguments in the appropriate registers, including the syscall number, and jumps to a syscall instruction (`syscall`/`int 0x80` on x86 Linux). However, CFI presents new challenges: Even the most coarse-grained CFI policy requires that indirect calls target the starts of functions, disallowing a jump to a syscall instruction in the middle of a function. In the C standard library, syscalls are made available through wrapper functions like `execve()` and `open()`, along with a general-purpose `syscall()`. While an attacker could instead jump to one of these wrapper functions, as was done in COOP [21], this can be prevented by more precise CFI: If syscall wrappers are marked as illegal targets for indirect calls, the only way for an

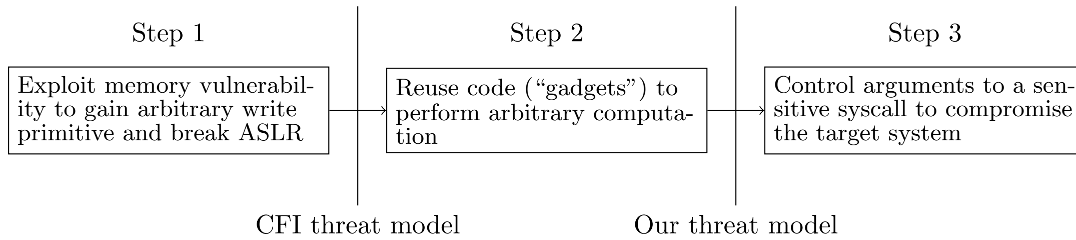

Figure 3.1: Steps in an abstracted end-to-end memory exploit. CFI defences typically assume that an attacker has already completed step 1 (i.e., found and triggered a memory bug), and seek to prevent step 2. We instead assume the attacker has already completed step 2 (i.e., performed arbitrary computation in user mode), and examine the effectiveness of CFI at preventing step 3.

attacker to trigger a syscall is through a wrapper call site in the binary itself. This significantly reduces the attack surface, as the attacker must now find a gadget which allows them to control the arguments to a syscall wrapper without violating the CFI policy. For instance, if a program only calls `exec` with the first argument hard-coded to `"/bin/ls"`, then the attacker can not use it to execute any other program. We describe our modelled CFI policies and possible enforcement mechanisms in §3.3. If no gadget for calling a syscall wrapper exists (e.g. if the program makes no syscalls deemed sensitive), then the attacker is trapped in the process sandbox despite having Turing-complete computation ability.

We therefore ask: *Over a variety of CFI policies, how hard it is to perform step 3?* To answer this, we consider an artificially strong attacker who has already succeeded in step 2 — that is, one who has arbitrary control over (writable) memory and may jump to any location allowed by the CFI policy. We identify sensitive syscall wrappers and find their call sites in the binary. If the attacker is able to control an argument to one of these call sites, we call that call site *exploitable* and the corresponding code path a *syscall gadget*. The existence of a syscall gadget does not guarantee that an actual attack can be carried out: Rather, the absence of such a gadget guarantees that an attack can not cause serious damage, since it represents the critical last step in the exploitation pipeline. We design our analysis to make conservative assumptions when necessary, with the intention of over-approximating the set of syscall gadgets to find an upper bound on the *exploitability* of the program. In other words, it is primarily intended to inform defences rather than generate attacks, although it could be useful for either if made sufficiently precise. By sweeping the level of CFI precision over a suite of programs, we examine factors that lead to exploitability and the effectiveness of CFI at mitigating it.

In short, our analysis consists of the following components:

1. A model of a strong attacker who can set writable memory and data registers to an arbitrary state at some point in time and jump to any location allowed by the CFI policy (§3.2);
2. a model of CFI the attacker is subject to, including a fully precise backward edge and varying policies for the forward edge (§3.3);
3. a static taint analysis for finding data flows from attacker-controlled variables to sensitive syscall arguments (§3.4).

3.2 Threat Model

We adopt a strong threat model in which the attacker has already achieved Turing-complete execution, and want to determine whether this attacker can control an argument to a sensitive syscall. We make standard assumptions about process isolation: With only access to the unprivileged ISA and having yet to issue a sensitive syscall, the attacker can not bypass page permissions or virtual memory, overwrite code pages, or execute data pages (NX). For calls into shared libraries, the integrity of the GOT/PLT is guaranteed through relocation read-only [15]. This causes the GOT/PLT to be populated eagerly at load-time and its pages set to read-only; it would otherwise be writable to support lazy binding and therefore corruptible by our attacker.

We impose varying CFI policies on the attacker, which are described in §3.3. If a function is the legal target of an indirect call (from any call site) according to the CFI policy, then the attacker is allowed to jump to it. Since the attacker has Turing-complete computation ability, they can set writable memory and data registers to an arbitrary state before jumping; once the attacker jumps, however, they do not regain control until the function returns.

As an example, consider the code snippet in Listing 3.1 and suppose that `foo` is a legal indirect target. The attacker controls globals and function parameters at the start of the function (`str1`, `str2`, and `flag`), but these values can only flow to certain sensitive syscalls and only if they are not overwritten before the call is made. We aim to identify the syscall arguments that can be controlled by such an attacker using static analysis.

```
1  char *str1;
2
3  // legal indirect target
4  void foo(char *str2, int flag) {
5      if (flag) {
6          sensitive(str1);    // exploitable
7          sensitive("hello"); // not exploitable
8      }
9      else {
10         sensitive(str2);    // exploitable
11         str2 = "goodbye";
12         sensitive(str2);    // not exploitable
13     }
14     sensitive(str2);        // exploitable if flag != 0
15 }
```

Listing 3.1: A toy program illustrating attacker capabilities. If `foo` is a legal indirect target according to the CFI policy, then `str1`, `str2`, and `flag` are considered attacker-controlled by line 4. These values flow to the sensitive calls on lines 6 and 10. However, if the argument is a constant (line 7) or locally overwritten (line 12), the attacker can not control it. Lastly, the sensitive call on line 14 is exploitable by the attacker if `flag` is not 0, and is overwritten otherwise.

The set of sensitive syscalls is configurable and depends on the modes of attack under consideration. In our work, we constrain ourselves to Linux and choose a set based on prior work that can lead to privilege escalation (§2.3):

- `execve` and `execveat` can execute arbitrary programs;

- `mmap`, `mremap`, `remap_file_pages`, and `mprotect` can change page permissions to overwrite code or execute data;
- `open` and `openat` can open files for editing, while `write` and variants can write to them; filesystem access is often as good as a shell (e.g. writing to `.ssh/authorized_keys` [6]).

3.3 CFI Model

We impose a variety of CFI policies on the attacker. For the backward edge, we assume full precision, which can be provided by a shadow stack — this rules out ROP entirely because return addresses can not be corrupted without detection. For the forward edge, we model four different policies, enforced at each indirect call:

- *any*: Indirect calls must target a function
- *addr*: Indirect calls must target an address-taken function
- *type*: Caller and callee type signatures must match
- *addr-type*: Caller and callee type signatures must match, and callee must be address-taken

The lowest-precision policy, *any*, ensures that an attacker can not trigger a syscall by jumping directly to a syscall instruction. *addr* and *type* are more precise than *any*, and *addr-type* is more precise than the other three. *type* and *addr-type* are called *call-site-sensitive* because the set of legal targets depends on the indirect call site. Conversely, *any* and *addr* are called *call-site-insensitive* because the set of legal targets is the same for every indirect call site. As discussed in §3.1, we additionally assume that syscall wrappers are not legal indirect call targets. While this could theoretically break some programs, we believe that indirect calls to syscall wrappers are uncommon and unnecessary; we also do not encounter any such uses in our evaluation.

Many mechanisms exist for enforcing CFI in both software and hardware; we give one possible mechanism for forward-edge CFI similar to the one in the original proposal [2] to serve a model. Call-site-insensitive policies are enforced by inserting a unique id into the binary before every function. A global read-only bitmap indicates if that function is a legal indirect call target. At each indirect call, the id is read by loading at a fixed offset from the call target; the call is only made if the the bitmap reads 1 when when indexed with the id. Call-site-sensitive policies are similar, except they also require an id for each call site. The bitmap in this case accepts two indicies (caller and callee ids), and reads 1 if the caller-callee pair is legal.

3.4 Syscall Reachability

In order to find syscall gadgets under our threat model, we use a specialization of static taint analysis which we call *syscall reachability analysis*. In this analysis, a tainted variable is one that may be controlled by the attacker. Per the threat model (§3.2), these variables initially consist of writable memory and data registers at any legal indirect target allowed by the CFI policy (§3.3). We then consider any variable derived from an attacker-controlled variable to also be attacker-controlled. This is an over-approximation since we do not explore the program's value space. Our definition of taint allows us to use standard data-flow analyses which assume correct program behaviour to reason about an attacker who has gained control of memory through undefined behaviour. This assumes

that the attacker does not also trigger undefined behaviour in the syscall gadget itself, which we discuss further in §3.5.

To perform syscall reachability analysis, we start by constructing a whole-program data-flow graph, where nodes are variables in the program and edges represent def-use relations. For instance, in Listing 3.1, the argument `str2` on line 12 would be a node, and its definition would be `"goodbye"` on line 11. Resolving edges for loads and stores in such a graph requires a whole-program points-to analysis.

We then identify, in the language of taint analysis, source nodes (attacker-controlled values) and sink nodes (arguments to sensitive syscalls). Source nodes are marked as *tainted*; all other nodes are initially *untainted*. To find flows from sources to sinks, we propagate taint along the data-flow graph, which is simply a graph reachability problem: If node A is tainted, its taint is propagated to node B iff there exists a path from A to B in the graph. This problem can be solved with a worklist algorithm, which is guaranteed to converge because each node is visited at most once (when it is first tainted).

Put more precisely, the analysis is defined by the following rules:

1. **Sources:** Indirect call targets allowed by CFI are taint sources. To find these, we find all indirect call sites (ICSs), find the set of legal targets at each ICS according to the CFI policy, and take the union of all of them. For each of these source functions, the function parameters and global variables visible at the start of the function are tainted.
2. **Sinks:** Sensitive syscall call sites are taint sinks. For each of these call sites, the function arguments are initially untainted; if taint is propagated to one of them, the call site is considered to be exploitable by an attacker.
3. **Propagation:** If a variable is defined by an expression that uses a tainted variable, it is also tainted.

We do not propagate taint through indirect calls, since the targets of these calls would be taint sources (and therefore already tainted). In other words, within a syscall gadget, we only consider data-flow paths through direct control-flow. In addition to simplifying the analysis, this rule allows us to prune irrelevant parts of the program to improve analysis scalability. We discuss this further in our implementation (§4.2).

As an example, consider again Listing 3.1: Since `foo` is an legal indirect call target, it is a source function. Thus, `str1`, `str2`, and `flag` are tainted; this taint propagates to `str1` on line 6 and `str2` on line 10, as well as `str2` on line 14 since we do not explore the value space of `flag`. `"hello"` on line 7 and `str2` on line 12 are untainted at the end of the analysis and declared safe.

If all arguments to a sink call site are untainted under this conservative analysis, we conclude that it is safe from exploitation. If an argument is tainted, the analysis can provide a data-flow path from source to sink. This path may be exploitable by an attacker, but it may also be a false positive for reasons we discuss in §3.5. By assigning IDs to sources and sinks, we can track sinks tainted by each source and sources tainting each sink, along with corresponding paths. This highlights sensitive points in the program and provides a set of metrics for evaluating the exploitability of CFI-protected programs. For call-site-sensitive CFI, we can also find the set of sinks reachable from each ICS: This is the union of sinks reachable from a source, for each source that is a legal target of that ICS under the CFI policy. This could be of interest if modelling a more limited attacker who only has access to certain ICSs, or perhaps for targeted exploits.

3.4.1 Modular Analysis for Shared Libraries

The analysis described above is intended for statically linked binaries. If a client program is dynamically linked against a library, the two can be analyzed independently, but this misses execution paths where the client calls into the library. For libc, we address this by creating an explicit list of wrapper functions that are capable of making each sensitive syscall. This idea can be extended to libraries to support a modular analysis:

1. Get a list of all function signatures in the library;
2. Analyze the client with the library functions as sinks, keeping track of the tainted functions;
3. Analyze the library with the tainted functions from the previous step as sources.

This has the added benefit of highlighting sensitive points at the well-defined library interface. It can be extended to an arbitrary DAG of dependencies by analyzing each binary in topological order. The major assumption is that the attacker begins in the client and calls into the library; the analysis misses control-flow paths that go the other way (e.g. the library calling a callback function provided earlier by the client). We note also that this modular analysis is orthogonal to independent analysis of the client and library: The former considers paths from client to library, while the latter two consider paths that begin and end in the respective binaries.

3.5 Sources of Imprecision

In order to make static analysis tractable, simplifying assumptions are required. We discuss here some assumptions made in our analysis and their possible impact on its results. Since our objective is to over-approximate the set of syscall gadgets under a strong attacker model, we generally make conservative assumptions, leading to potential false positives:

1) The underlying pointer analysis is conservative. In order to construct a whole-program data-flow graph, we require an underlying points-to analysis, which is challenging to do precisely and undecidable in general. Over-approximation in points-to sets can lead to spurious dataflows where a particular load instruction is claimed to load a tainted variable out of memory, despite it not actually being possible to load that variable. Improving the precision of points-to analysis is an active area of research which we consider to be orthogonal to our work.

2) Static taint analysis is conservative. While dynamic taint tracking propagates taint along the executed control-flow path, static taint analysis must make conclusions that are true of all possible paths, leading to over-approximation. Without path-sensitivity, it allows infeasible paths, such as entering a function from one call site and exiting through another.

Despite our effort to keep error one-sided, using taint analysis to represent attacker-controlled values requires two assumptions which may lead to false negatives:

3) Undefined behaviour in syscall gadgets is ignored. Our use of data-flow analysis assumes correct program behaviour within the syscall gadget. That is, while the attacker is assumed to have exploited a memory bug to perform arbitrary computation before calling the syscall gadget, we

assume there is no further memory misuse in the syscall gadget itself. This is not guaranteed in reality, so we may miss an exploitable syscall.

4) Tainted control flow is ignored. As a standard assumption in taint analysis, we do not consider branch conditions part of the def-use chain. For example, in the statement `ch = cond ? 'a' : 'b';`, if `cond` is tainted, that taint does not propagate to `ch` because it is only derived from `'a'` and `'b'`. In theory, this means we could miss an attacker who influences a syscall argument through branch conditions alone, but it is unclear what code patterns would enable this; we leave this to future work.

Chapter 4

Implementation

To perform syscall reachability analysis, we implemented a prototype tool using the LLVM toolchain and SVF [28] analysis framework. This chapter describes the tool's workflow (§4.1) and scalability optimizations (§4.2).

4.1 Workflow

Our prototype takes as input a whole-program LLVM bitcode file, and internally produces two bitmaps representing (1) the set of sources (indirect target functions) that can be called at each indirect call site (ICS) according to the CFI policy, and (2) the set of sinks (sensitive syscall call sites) that are tainted by each source. As discussed in §3.4, these two can be combined to find the set of sinks exploitable from each ICS. The prototype has a library and client component: The library gives direct access to the bitmaps, while the client is a command-line tool which produces a report on tainted syscall call sites along with summary statistics. The tool operates in three stages: (1) Finding sources and sinks, (2) propagating taint, and (3) reporting results. These stages are described in more detail below.

1) Find sources and sinks. Taint sources are legal indirect call targets and taint sinks are sensitive syscall call sites. We support the CFI policies described in §3.3 for finding sources: The tool first finds all ICSs, and then applies the desired CFI policy at each ICS to find to find the set of legal targets (creating bitmap 1). The total set of sources is the union of legal targets over all ICSs. In order to implement a modular client-library analysis (§3.4.1), the tool also allows a user-specified list of source function names or signatures to be used instead.

To find sinks, the tool finds call sites of functions that wrap the syscalls listed in §3.2. Each syscall has a libc wrapper with the same name, and often several variants. We produced this list of sensitive syscall wrapper names manually, and while we feel it is adequate for a prototype, a missing function could potentially lead to false negatives in our results. This could be addressed by analyzing an implementation like glibc to find all functions issuing syscall instructions; we leave this to future work. The full list is quite long — we list a few of the variants here for reference:

- `execve`: `execv`, `execvp`, `execl`
- `mmap`: `mmap64`

- `open`: `open64`, `fopen`, `ofstream::ofstream`
- `write`: `fputs`, `fprintf`, `ostream::operator<<`

Since C++ stream objects are templated, each template instantiation needs to be listed separately: We currently just support `char *` and `std::basic_string<char>`.

2) Build SVFG and propagate taint. In order to propagate taint, the tool uses SVF to construct a sparse value-flow graph (SVFG), which is a whole-program data-flow graph. SVF resolves points-to sets at LLVM loads and stores, and thus requires a whole-program points-to analysis. This represents a scalability bottleneck in both time and memory [29]. To alleviate this, our prototype searches the CFG to find pointers relevant to source-sink paths, and applies the points-to analysis to only these pointers. We call this *CFG pruning*; it is described in more detail in §4.2.

Once the SVFG is created, the tool finds source and sink nodes. For each source function, it finds the `FormalParm` nodes corresponding to function parameters and `FormalIN` nodes corresponding to memory objects visible at the start of the function; for each sink call site, it finds the `ActualParm` nodes corresponding to function arguments.

Taint is propagated by finding all reachable nodes on the SVFG using a worklist algorithm (breadth-first search). The tool supports propagating taint either forward (finding all nodes reachable from a source) or backward (finding all nodes that can reach a sink). This yields the set of sinks tainted by each source (bitmap 2). The tool also keeps a depth field and back pointer for each node during taint propagation to efficiently find the shortest tainting path. In practice, we find that most programs have far fewer sinks than sources, so backward propagation is usually faster than forward.

3) Report results. After taint propagation is done, The command line tool lists summary statistics, the states of all sinks (tainted or untainted), and source-sink data-flow paths. Since the tool just summarizes the two bitmaps, its output is easily configurable. By default, it prints the path from the nearest source node to each tainted sink node.

4.2 Control-Flow Graph Pruning

Since a SVFG resolves the points-to sets of LLVM loads and stores, it requires a points-to analysis. The most basic choice is an inclusion-based (Andersen's) analysis, which has worst-case cubic time complexity (realistically, quadratic [26]), limiting our ability to analyze large programs in practice [29].

To ameliorate this, our prototype uses *CFG pruning* to avoid analyzing pointers that do not affect the correctness of its analysis. The idea is that since the attacker is assumed to control all writable memory and function parameters at the start of a source function, pointers which are not defined on any source-sink control-flow path do not need to be analyzed. Since these pointers must have been allocated before jumping to a source function (if allocated at all), any capability they grant the attacker is one that the attacker already has through arbitrary control over memory.

To find reachable basic blocks, the tool searches the interprocedural control-flow graph (ICFG): If propagating taint forward, it starts from all sources and searches forward; if propagating taint backward, it starts from all sinks and searches backward. Since we assume a shadow stack is present (§3.3), we do this CFG traversal in a context-sensitive way where a function return must return

to the caller; without this, a naive search would allow returning to any other call site and over-approximate the set of reachable basic blocks. Additionally, in our threat model, since we are only interested in direct control-flow within a syscall gadget (§3.4), we do not need to resolve indirect call targets in the ICFG. This avoids a chicken-and-egg problem where adding indirect edges to the ICFG which would require its own points-to analysis.

After finding the set of reachable basic blocks B, the tool also finds the set of functions F containing all basic blocks in B. It then prunes the set of pointers to be analyzed according to the following rules:

1. If the pointer is defined in a basic block b (e.g. by an instruction), keep it if $b \in B$ and discard it otherwise;
2. If the pointer is not defined in a basic block but is defined in a function f (e.g. as a function parameter), keep it if $f \in F$ and discard it otherwise;
3. If the pointer is not defined in a basic block or function (e.g. as a global), keep it.

The tool prunes these pointers from SVF's *constraint graph*. After running the points-to analysis, the resulting SVFG contains all pointers in the original program, but the ones that were pruned have no points-to targets.

We originally also supported a serial CFG pruning mode where the tool would build a SVFG, propagate taint, and deallocate the SVFG individually for every source or sink, with the expectation that it might be slower but would use less memory. It actually used more memory, which we speculate was due to either a memory leak or the kernel page allocator deferring reclamation, and also became extremely slow compared to the other options as we tested larger programs with an increasing number of sinks.

Chapter 5

Evaluation

Our experiments are motivated by the following evaluation questions:

EQ1: How often are programs considered exploitable against our strong threat model over varying CFI policies (§3.2)?

EQ2: What code patterns around sensitive syscalls lead to exploitability, and can this weakness be mitigated?

EQ3: How realistic are the data-flow paths discovered by the taint analysis? What mistakes does it make due to imprecision (§3.5)?

EQ4: Does the analysis scale well to large programs?

To address these questions, we ran our analysis tool on a suite of popular programs written in C and C++. The following sections describe our test setup (§5.1), quantitative (§5.2) and qualitative (§5.3, §5.4) evaluation of exploitability, and scalability measurements (§5.5).

5.1 Experiment Setup

We ran our analysis on an Intel Xeon Gold 5218 processor with 512GB of memory running Ubuntu 22.04. The programs tested are shown in Table 5.1. We chose these programs based on popularity, test set diversity, perceived value as an exploit target, difficulty of build configuration, use in prior work for comparison, and with the constraint that a sigificant portion be written in C or C++.

Each program was compiled with Clang 13.0.1-2ubuntu2, using gllvm [30] as a wrapper to link together a whole-program bitcode file.

Project	Version	Language	SLOC	Program	Description
Firefox	102.0.1	C, C++, *	9.26M	`firefox`	Web browser
				`updater`	Firefox updater
				`crashreporter`	Firefox crash GUI
Apache	2.4.53	C	309k	`httpd`	Web server
	1.7.0		64.1k	`libapr-1.so`	Apache library
NGINX	1.20.2	C	141k	`nginx`	Web server
DBMail	3.2.6	C	65.0k	`dbmail-imapd`	Mail server
				`libdbmail.so`	DBMail library
PostgreSQL	14.3	C	879k	`psql`	Relational database
LevelDB	1.23	C++	20.9k	`leveldbutil`	Key-value store
Wireshark	3.6.5	C, C++	2.92M	`wireshark`	Packet analyzer
				`tshark`	Wireshark CLI
				`libwireshark.so`	Wireshark library
				`libwiretap.so`	Wireshark logging library
OpenVPN3*	v17_beta	C++	310k	`openvpn3`	VPN client GUI
				`openvpn3-admin`	OpenVPN3 config tool
Xpdf	4.04	C++	128k	`xpdf`	PDF viewer

Table 5.1: Test program suite, indicating project size, version, and associated binaries. Sizes in SLOC were measured using SLOCCount [25], and only include C and C++. Notes: (1) The Firefox codebase contains several other languages, including JavaScript, Python, and Rust; (2) OpenVPN3 here is the openvpn3-linux client GUI, which links against the openvpn3 library.

5.2 Summary Statistics

In order to assess the exploitability of common programs under various levels of CFI protection (EQ1), we ran our analysis tool on the test suite. As described in section §4.1, the CFI policy determines the set of functions treated as taint sources.

Summary statistics from the test run are shown in Table 5.2. For each CFI policy, we count the number of sources tainting a sink, and sinks tainted by a source. Each sink is a call site of a sensitive syscall (§3.4), which may be used to cause damage to the system. We group sinks by the capability the syscall grants the attacker:

- *exec* for running arbitrary programs, including `execve` and `execveat`;
- *mmap* for modifying page permissions, including `mmap`, `mremap`, `remap_file_pages`, and `mprotect`;
- *open* for opening a file, including `open` and `openat`;
- *write* for writing to a file through a file descriptor, including `write`, `pwrite`, and `writev`;
- *syscall* for the libc `syscall`, which allows a syscall to be called by number.

All analyses completed within a few minutes, except for `libwireshark.so` on *open* syscalls, which we stopped after 2 hours. Details on time and memory use are given in §5.5.

Program	CFI	Sinks (tainted/total)					Sources
		exec	mmap	open	write	syscall	(tainting/total)
firefox	any	0/0	3/4	7/12	6/47	20/230	651/1,405
	addr	0/0	0/4	1/12	1/47	16/230	103/260
	type	0/0	0/4	4/12	1/47	6/230	64/162
	addr-type	0/0	0/4	1/12	1/47	6/230	43/90
updater	any	1/1	0/0	24/28	0/0	0/0	26/342
	addr	0/1	0/0	4/28	0/0	0/0	5/55
	type	0/1	0/0	1/28	0/0	0/0	2/7
	addr-type	0/1	0/0	1/28	0/0	0/0	1/3
crashreporter	any	1/3	0/0	3/18	0/44	0/0	264/720
	addr	1/3	0/0	2/18	0/44	0/0	30/102
	type	1/3	0/0	2/18	0/44	0/0	27/64
	addr-type	1/3	0/0	2/18	0/44	0/0	12/29
httpd	any	0/0	0/0	1/1	0/2	0/1	1/2,068
	addr	0/0	0/0	1/1	0/2	0/1	1/1,865
	type	0/0	0/0	1/1	0/2	0/1	1/576
	addr-type	0/0	0/0	1/1	0/2	0/1	1/548
libapr-1.so	any	4/5	2/3	5/6	15/15	0/0	318/902
	addr	0/5	0/3	1/6	5/15	0/0	32/101
	type	0/5	0/3	4/6	5/15	0/0	34/142
	addr-type	0/5	0/3	1/6	5/15	0/0	27/87
nginx	any	1/1	2/2	20/22	18/22	0/1	904/1,359
	addr	1/1	1/2	18/22	16/22	0/1	573/759
	type	1/1	1/2	18/22	14/22	0/1	374/522
	addr-type	1/1	1/2	18/22	13/22	0/1	325/446
dbmail-imapd	any	0/0	0/0	0/0	117/117	0/0	29/402
	addr	0/0	0/0	0/0	117/117	0/0	29/92
	type	0/0	0/0	0/0	0/117	0/0	0/58
	addr-type	0/0	0/0	0/0	0/117	0/0	0/46
libdbmail.so	any	0/0	1/1	8/9	2/3	0/0	376/1,048
	addr	0/0	1/1	4/9	1/3	0/0	33/63
	type	0/0	1/1	2/9	1/3	0/0	81/172
	addr-type	0/0	1/1	0/9	1/3	0/0	6/15
psql	any	1/3	0/0	12/17	2/3	0/0	324/837
	addr	1/3	0/0	11/17	2/3	0/0	42/237
	type	1/3	0/0	11/17	0/3	0/0	56/281
	addr-type	1/3	0/0	11/17	0/3	0/0	29/223
leveldbutil	any	0/0	1/1	0/9	2/7	0/0	34/1,325
	addr	0/0	1/1	0/9	1/7	0/0	9/200
	type	0/0	0/1	0/9	1/7	0/0	1/97
	addr-type	0/0	0/1	0/9	1/7	0/0	1/70

Program	CFI	Sinks (tainted/total)					Sources
		exec	mmap	open	write	syscall	(tainting/total)
wireshark	any	1/2	0/0	14/33	5/5	0/0	1,349/13,961
	addr	0/2	0/0	9/33	1/5	0/0	521/5,561
	type	0/2	0/0	5/33	4/5	0/0	502/3,191
	addr-type	0/2	0/0	3/33	1/5	0/0	218/1,858
tshark	any	1/2	0/0	0/5	4/4	0/0	5/1,063
	addr	0/2	0/0	0/5	0/4	0/0	0/254
	type	0/2	0/0	0/5	0/4	0/0	0/180
	addr-type	0/2	0/0	0/5	0/4	0/0	0/99
libwireshark.so	any	0/0	0/0	–	1/1	0/0	*1/79,351
	addr	0/0	0/0	–	1/1	0/0	*1/70,869
	type	0/0	0/0	–	1/1	0/0	*1/72,140
	addr-type	0/0	0/0	–	1/1	0/0	*1/70,546
libwiretap.so	any	0/0	0/0	5/6	1/1	0/0	89/1,111
	addr	0/0	0/0	2/6	1/1	0/0	63/451
	type	0/0	0/0	0/6	1/1	0/0	55/486
	addr-type	0/0	0/0	0/6	1/1	0/0	55/421
openvpn3	any	0/0	0/0	1/1	207/1,210	0/0	5,383/13,770
	addr	0/0	0/0	0/1	121/1,210	0/0	509/1,655
	type	0/0	0/0	0/1	94/1,210	0/0	653/1,434
	addr-type	0/0	0/0	0/1	94/1,210	0/0	315/701
openvpn3-admin	any	0/0	0/0	2/2	53/311	0/0	447/1,928
	addr	0/0	0/0	1/2	28/311	0/0	24/212
	type	0/0	0/0	1/2	28/311	0/0	28/117
	addr-type	0/0	0/0	1/2	28/311	0/0	15/80
xpdf	any	1/1	0/0	23/23	0/0	0/0	1,974/5,674
	addr	0/1	0/0	19/23	0/0	0/0	523/1,889
	type	1/1	0/0	18/23	0/0	0/0	580/1,601
	addr-type	0/1	0/0	18/23	0/0	0/0	401/1,229

Table 5.2: Summary statistics for binaries. *Program* indicates the program name, *CFI* indicates the CFI policy used to find taint sources, *Sinks* indicates the number of tainted (reached by a source) and total sink call sites, and *Sources* indicates the number of tainting (reaching a sink) and total source functions. Four CFI policies are tested: *any* (all functions are sources), *addr* (address-taken functions are sources), *type* (caller-callee type signature matching), and *addr-type* (intersection of addr and type). Sinks are grouped by capability: *exec* indicates program launching; *mmap* indicates page permission modification; *open* and *write* indicate filesystem access and modification, respectively; *syscall* indicates libc `syscall`. For `libwireshark.so`, the analysis on *open* syscalls timed out; the number of tainting sources is measured with respect to the other four types of syscalls.

Overall, we find that most programs make relatively few sensitive syscalls, even in large codebases. There are a few outliers (e.g. *syscall* in `firefox` and *write* in `dbmail-imapd`, `openvpn3`, and `openvpn3-admin`), which we explore later in more detail (§5.3). Many sinks are untainted even when

the CFI policy is set to *any*: The simplest way this can happen is when all the arguments to a syscall are constants (e.g. opening a file with a fixed name, or writing a fixed message to stdout). We also see a clear decrease in exploitability as the precision of CFI is increased: More precise CFI means fewer legal targets, i.e., fewer source functions. With fewer source functions, fewer sink call sites are tainted by those sources. We see this drop in total sources and tainted sinks across all programs when comparing the *addr-type* policy to the *any* policy. With the *addr-type* policy, for our tested set of sensitive syscalls, two programs in our test suite would be considered unexploitable because they have no tainted sinks at all (`dbmail-imapd` and `tshark`). Furthermore, the less precise *addr* policy is often comparable in tainted sinks to *addr-type* (e.g. `firefox`, `tshark`) despite being significantly cheaper to enforce.

Across programs, there is signficant variation in both the types of sinks that are present and the fraction of sinks that are tainted. Some of this can be explained by function: `xpdf` is a read-only pdf viewer, so it does not write to the filesystem at all and therefore has no *write* sinks. In other cases, however, the connection between the type of program and set of sensitive syscalls used is less clear.

The actual number of sinks is sensitive to compiler optimization and code structure: If the function a sink is in gets inlined, each caller of that function now has a sink. This inflates the number of sinks but is actually good for analysis precision, since it essentially gives us context-sensitivity. Also, since this experiment only considered individual binaries, syscalls outside of the binary were not analyzed. For instance, while `httpd` has relatively few sinks, much of its functionality is factored into `libapr.so`. Furthermore, The analysis of `libapr.so` does not consider an attacker who compromises the client and calls into the library. To address this, we repeated the experiment for shared libraries, using a client program to find reachable entry points as described in §3.4.1. Results from this run are shown in Table 5.3.

In analyzing libraries through clients, we find that there are fewer sources and tainted sinks than when the library is analyzed on its own with the *any* policy. This is expected, as only the functions in the library reachable by the client are considered sources, instead of all of them. Comparing against other CFI policies, the results vary: `httpd/libapr-1.so` has more sources and tainted sinks over all modes than `libapr-1.so` in modes other than *any*; an opposite trend exists when comparing `tshark/libwiretap.so` to `libwiretap.so`. As discussed in §3.4.1, these analyses are orthogonal: Analyzing the library finds syscall gadgets contained entirely in the library, while analyzing the library through a client finds syscall gadgets that begin in the client and end in the library.

We do not see much variation in sources and tainted sinks in the library across CFI precision levels in the client, other than when comparing *any* against the other policies. We speculate that this is related to how libraries are used: Perhaps if a client uses a library function once, it is likely to use it multiple times, leading to most library functions that the client does use becoming sources in this analysis; on the other hand, library functions that are never used by the client are excluded.

5.2.1 Indirect Call Site Analysis

While we consider a sink (syscall call site) to be exploitable if it is tainted by any source (indirect target function), not every source can be called at every indirect call site if the CFI policy is call-site-sensitive. As discussed in §3.4, we can find the set of exploitable sinks for each ICS by (1) finding the set of source functions that can be called at that ICS according to the CFI policy, and

Program (client/library)	**CFI**	**Sinks** (tainted/total)					**Sources** (tainting/total)
		exec	mmap	open	write	syscall	
`httpd/` `libapr-1.so`	any	4/5	1/3	5/6	11/15	0/0	126/233
	addr	4/5	1/3	5/6	9/15	0/0	124/226
	type	4/5	1/3	5/6	9/15	0/0	110/198
	addr-type	4/5	1/3	5/6	9/15	0/0	110/198
`dbmail-imapd/` `libdbmail.so`	any	0/0	1/1	1/9	2/3	0/0	94/158
	addr	0/0	1/1	0/9	2/3	0/0	92/148
	type	0/0	1/1	0/9	2/3	0/0	41/60
	addr-type	0/0	1/1	0/9	2/3	0/0	38/57
`tshark/` `libwireshark.so`	any	0/0	0/0	–	0/1	0/0	*0/237
	addr	0/0	0/0	–	0/1	0/0	*0/176
	type	0/0	0/0	–	0/1	0/0	*0/119
	addr-type	0/0	0/0	–	0/1	0/0	*0/100
`tshark/` `libwiretap.so`	any	0/0	0/0	4/6	1/1	0/0	4/98
	addr	0/0	0/0	1/6	0/1	0/0	1/75
	type	0/0	0/0	1/6	0/1	0/0	1/55
	addr-type	0/0	0/0	1/6	0/1	0/0	1/50

Table 5.3: Summary statistics for client-library analysis (§3.4.1). Headings are as in Table 5.2, with a few caveats: (1) *CFI* refers to the CFI policy used to analyze the client, (2) *Sinks* refers to sinks in the library, and (3) *Sources* refers to functions in the library that can be called by an attacker through the client. As before, `libwireshark.so` timed out on *open* syscalls.

(2) taking the union of the sinks tainted by each of those sources. This adds a new dimension to exploitability: Since a real attacker may not be able to access every indirect call site, a sink could be *more* exploitable if it can be exploited from more ICSs. Similarly, an ICS could be more dangerous if more sinks can be exploited from it.

We measured this *distribution* of tainted sinks over ICSs on the test suite by finding the median and maximum number of sinks exploitable from each ICS (using the two call-site-sensitive CFI policies). The results for each sink type are shown in Table 5.4. These numbers are bounded above by the total number of tainted sinks from Table 5.2, which can be thought of as the union of exploitable sinks over all ICSs.

For all programs and all sinks other than *open* on `wireshark`, *max* is close to *union*. That is, there is at least one ICS which can be used to exploit almost any sink that can be exploited at all (from any ICS). On the other hand, *med* is often zero, meaning that more than half of ICSs can not be used to exploit any sink. This is true even when *max* is not zero – that is, the distribution of tainted sinks over ICSs has a long tail. There are a few exceptions to this pattern, particularly `nginx` and `psql`. In these programs, if a sink can be exploited, then it can be exploited from almost any ICS. The number of tainted sinks for these programs is also close to the total number of sinks, suggesting that the sinks may simply be reachable from a large portion of the program.

Program	CFI	Sinks (med/max/union/total)				
		exec	mmap	open	write	syscall
`firefox`	type	0/0/0/0	0/0/0/4	0/3/4/12	0/1/1/47	0/6/6/230
	addr-type	0/0/0/0	0/0/0/4	0/1/1/12	0/1/1/47	0/6/6/230
`updater`	type	0/0/0/1	0/0/0/0	1/1/1/28	0/0/0/0	0/0/0/0
	addr-type	0/0/0/1	0/0/0/0	1/1/1/28	0/0/0/0	0/0/0/0
`crashreporter`	type	1/1/1/3	0/0/0/0	2/2/2/18	0/0/0/44	0/0/0/0
	addr-type	0/1/1/3	0/0/0/0	0/2/2/18	0/0/0/44	0/0/0/0
`httpd`	type	0/0/0/0	0/0/0/0	0/1/1/1	0/0/0/2	0/0/0/1
	addr-type	0/0/0/0	0/0/0/0	0/1/1/1	0/0/0/2	0/0/0/1
`libapr-1.so`	type	0/0/0/5	0/0/0/3	0/4/4/6	0/5/5/15	0/0/0/0
	addr-type	0/0/0/5	0/0/0/3	0/1/1/6	0/5/5/15	0/0/0/0
`nginx`	type	0/1/1/1	1/1/1/2	16/18/18/22	11/12/14/22	0/0/0/1
	addr-type	0/1/1/1	1/1/1/2	16/18/18/22	11/12/13/22	0/0/0/1
`dbmail-imapd`	type	0/0/0/0	0/0/0/0	0/0/0/0	0/0/0/117	0/0/0/0
	addr-type	0/0/0/0	0/0/0/0	0/0/0/0	0/0/0/117	0/0/0/0
`libdbmail.so`	type	0/0/0/0	1/1/1/1	0/1/2/9	0/1/1/3	0/0/0/0
	addr-type	0/0/0/0	0/1/1/1	0/0/0/9	0/1/1/3	0/0/0/0
`psql`	type	1/1/1/3	0/0/0/0	10/11/11/17	0/0/0/3	0/0/0/0
	addr-type	1/1/1/3	0/0/0/0	10/11/11/17	0/0/0/3	0/0/0/0
`leveldbutil`	type	0/0/0/0	0/0/0/1	0/0/0/9	0/1/1/7	0/0/0/0
	addr-type	0/0/0/0	0/0/0/1	0/0/0/9	0/1/1/7	0/0/0/0
`wireshark`	type	0/0/0/2	0/0/0/0	0/1/5/33	0/3/4/5	0/0/0/0
	addr-type	0/0/0/2	0/0/0/0	0/1/3/33	0/1/1/5	0/0/0/0
`tshark`	type	0/0/0/2	0/0/0/0	0/0/0/5	0/0/0/4	0/0/0/0
	addr-type	0/0/0/2	0/0/0/0	0/0/0/5	0/0/0/4	0/0/0/0
`libwireshark.so`	type	0/0/0/0	0/0/0/0	0/0/0/1	0/1/1/1	0/0/0/0
	addr-type	0/0/0/0	0/0/0/0	0/0/0/1	0/1/1/1	0/0/0/0
`libwiretap.so`	type	0/0/0/0	0/0/0/0	0/0/0/6	0/1/1/1	0/0/0/0
	addr-type	0/0/0/0	0/0/0/0	0/0/0/6	0/1/1/1	0/0/0/0
`openvpn3`	type	0/0/0/0	0/0/0/0	0/0/0/1	0/76/94/1,210	0/0/0/0
	addr-type	0/0/0/0	0/0/0/0	0/0/0/1	0/76/94/1,210	0/0/0/0
`openvpn3-admin`	type	0/0/0/0	0/0/0/0	0/1/1/2	0/25/28/311	0/0/0/0
	addr-type	0/0/0/0	0/0/0/0	0/1/1/2	0/25/28/311	0/0/0/0
`xpdf`	type	0/1/1/1	0/0/0/0	0/18/18/23	0/0/0/0	0/0/0/0
	addr-type	0/0/0/1	0/0/0/0	0/18/18/23	0/0/0/0	0/0/0/0

Table 5.4: Distribution of exploitable sinks. *Sinks* indicates the median and maximum number of sinks exploitable from each ICS (*med/max*), total exploitable sinks (*union*), and total sinks (*total*). The latter two are repeated from Table 5.2 for reference.

5.3 Syscall Usage

To better understand how sensitive syscalls are used and the patterns that lead to a syscall being tainted (EQ2), we looked through our analysis tool's output on the test suite and compared it against the corresponding source code. Some selected examples are given as case studies in §5.4. In this section, we summarize syscall uses by type, explore outliers, and discuss potential mitigations.

exec usage varies from program to program, and some do not use it at all. One use is to run auxiliary programs: `updater` is an update tool for Firefox which restarts `firefox` after updating it. `crashreporter` is a GUI that runs after `firefox` has crashed, and it uses exec to start a crash dump analyzer program. The tainted sink in `crashreporter` is a false positive likely due to points-to imprecision: The program takes the name of the dump analyzer and modifies the string to find the full path; the analysis claims a tainted variable in another file can be loaded during this process, which does not seem possible based on our manual inspection. `wireshark` and `tshark` support dumping packet traces to disk, and do this by starting a separate packet capture program. `nginx` supports live updates: The path to its executable is stored as a global (writable!) string. To update the software, the user switches out the binary on disk, and sends the `nginx` process a signal which causes it to exec itself. It supports this with an address-taken wrapper around exec (`ngx_execute_proc`), so its exec sink is tainted and exploitable. `xpdf`'s use seems to be dead code: There is an `executeCommand` function which wraps around `system`, but does not have any callers according to `grep`. `psql` is interesting: as a command-line interface to PostgreSQL, it runs arbitrary shell commands from the user. Intuitively, it would not be surprising for an attacker with arbitrary control over memory to be able to exploit such a program. However, our results are inconclusive: The prototype analysis only reports one tainted path, which is invalid: The corresponding control-flow path has more returns than calls. A possible future direction for this work would be validating tainting data-flow paths based on control flow. As a library, `libapr-1.so` provides general process creation functions for parallelism. It has a similar exec wrapper to `nginx` (`apr_proc_create`), which is tainted across all CFI policies in the client-library analysis. The one untainted exec sink is actually due to a SVF bug where the SVFG is missing edges because `memcpy` is not handled correctly, giving us a false negative.

For programs where the set of exec targets is small and fixed, a sensible defense is input sanitization: The program to be exec'd can be compared against a list of legal ones, and rejected if is not in the list. For instance, if `updater` only ever exec's `firefox` in benign execution, an attacker who tries to exploit it by running `/bin/sh` could be stopped by checking that the program argument to exec is, in fact, `firefox`. This can be done either through compiler instrumentation or by interposing on all syscalls with a reference monitor [4]. For libraries with "exec a program" functions where the set of possible targets is not known to the library, one solution is to treat the library as a black box and sanitize arguments passed to the library in the client. Again, this requires that the set of legal programs started by the client be known in advance.

mmap, while capable of changing page permissions, is overwhelmingly used for memory allocation. The address field is usually null (asking the kernel to allocate a new buffer somewhere), and the protection field is specified as a constant in every case we examined (e.g. `PROT_READ | PROT_WRITE`). When an argument is tainted, it is usually the length field; the pointer is tainted in a few cases. A

tainted pointer to mmap could be dangerous if the protection field allows writing or execution (even if this is a constant), since an attacker could modify just the pointer to cause mmap to make data executable or code writable.

By filtering out benign memory allocation, we can eliminate mmap sinks from most programs. For the rest, as before, simple input sanitization can be effective: The vast majority of programs have no need to mark pages as executable, so this protection flag can simply be disabled for them. Since the size of the binary is known in advance, attempts to make the code region writable can also be blocked by checking if the mmap pointer is in that range.

open is used for filesystem access. The protection fields are typically constants like with mmap, but the file paths are usually variable. Many of the tested programs include logging features and can dump to a configurable file location. Due to the large variety of filesystem use patterns, it is difficult to draw any general conclusions about what causes an open syscall to be tainted. The output from the tool is also difficult to parse for open, and we suspect many of the tainted paths to be invalid; however, the path provided by the tool is just one of many possible ones and it is infeasible to manually read them all.

Regardless, damage to the filesystem can be mitigated by providing more fine-grained access control. Processes created by a user do not need to be able to access every file that that user can: Sensitive data like cryptographic keys and lists of trusted users can be made isolated and only accessible by certain programs, reducing the attack surface.

write is used for writing to a file handle. While we are interested in call sites an attacker could use to modify a sensitive file, our output is polluted by writes to stdout and stderr. This is less of an issue in C, since writing to stdout is more commonly done with `printf` or `puts`. However, using `cout` in C++ calls `std::ostream::operator<<`, which is the same function used to write to any `std::ofstream`. Further, `openvpn3` and `openvpn3-admin` often take `std::ostream` objects as parameters to virtual functions (which are address-taken and therefore taint sources) and then write through them, causing the write sink to be tainted. An attacker could potentially exploit any of these call sites, although they would first have to open the desired file. `dbmail-imapd` opens a pipe to itself and writes a constant character to it to send an asynchronous notification. These writes are exploitable under the *any* and *addr* CFI policies because the file descriptor for the self pipe is loaded out of a non-constant global array, which is controlled by our attacker; under the other policies, there exists no control-flow path to trigger this write.

Compared to open syscalls, where a program is likely to have a fixed set of files it either would ever need or would never need to access, it is harder to define an access control policy for file descriptors. However, since a file must be opened before it can be written to, write syscalls arguably do not matter if the protection on open is adequate.

syscall is used to issue any syscall, typically one that does not have a dedicated libc wrapper. In every use of *syscall* in the test suite, the syscall number argument is a constant. Breaking down the 230 uses in `firefox`, we find:

- 206 inlined uses of `gettid`, 0 tainted;
- 11 uses of `mmap` in the memory allocator to bypass code that overrides libc mmap, 2 tainted;

- 7 uses of munmap for the same reason, 3 tainted;
- 3 uses of getrandom, 0 tainted;
- 3 uses of tgkill, 1 tainted.

The other two untainted uses are another gettid in httpd and a capset with constant arguments in nginx.

By identifying cases where the syscall number is constant, we can analyze the call site as we would any other call site of that syscall (e.g. syscall(SYS_mmap) as mmap()). While a program could allow the syscall number to be a variable, we feel there is no need to allow this, and it was not a problem for any program we tested. This can be enforced during the software development process and caught by a simple linter rule.

Overall, there are not very many syscall gadgets in most programs, and our manual analysis suggests that the harm caused by the ones that remain can be largely mitigated through lightweight input sanitization. What our analysis provides is a quantitative measure of exploitability (number of tainted sinks, for sensitive syscalls of interest) with a corresponding qualitative explanation (tainted source-sink path). It allows the defence to be tailored to the application, as input sanitization is only necessary for the syscall call sites that are considered exploitable. By modelling different levels of CFI precision, it allows the benefit of CFI to be assessed in the context of an end-to-end exploit, and on a program-by-program basis.

5.4 Case Studies

We now give a few illustrative code examples from the test suite and discuss the accuracy of the source-sink paths reported by our analysis tool (EQ3). In all cases, the CFI policy is *addr-type*.

5.4.1 NGINX

1) Exploitable exec in nginx. Listing 5.1 shows relevant source code for an exploitable exec in nginx. ngx_execute_proc is a legal indirect target, so the parameter data and anything it points to are tainted; arguments to execve are loaded out of it. Our tool correctly identifies this case: The tainted data-flow path is highlighted in yellow.

```
1  // os/unix/ngx_process.c:269
2  // legal indirect target; data is tainted
3  static void
4  ngx_execute_proc(ngx_cycle_t *cycle, void *data)
5  {
6      ngx_exec_ctx_t  *ctx = *data;
7
8      // all args are tainted
9      if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
10         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
11                       "execve() failed while executing %s \"%s\"",
12                       ctx->name, ctx->path);
13     }
```

```
14
15     exit(1);
16 }
```

Listing 5.1: Source code for an exploitable exec in `nginx`. `data`'s points-to set is tainted (line 4); it is copied to `ctx` (line 6) and loaded into `execve` (line 9).

2) False positive mmap in `nginx`. Listing 5.2 shows relevant source code for an `mmap` in `nginx` which the tool reports as tainted but is not actually exploitable. The tainted path is highlighted in yellow. `ngx_event_module_init` is a legal indirect target, so `cycle` and anything it points to are tainted. This taints `shm.log`, and then `shm.size` is passed as the length parameter to `mmap`. There is no actual data-flow path here, but the points-to analysis can not distinguish between `shm.log` and `shm.size` because it is not field-sensitive. Thus, it claims that `shm.size` is tainted by `cycle`. Also, even if an attacker could control `shm.size`, it would only determine the size of the mmap allocation; no permissions for existing pages would change.

```
1  // event/ngx_event.c:466
2  // legal indirect target; cycle is tainted
3  static ngx_int_t
4  ngx_event_module_init(ngx_cycle_t *cycle)
5  {
6      // ...
7      ngx_shm_t            shm;
8      // ...
9      // size is a constant
10     shm.size = size;
11     // ...
12     // w/o field-sensitivity, shm is tainted
13     shm.log = cycle->log;
14
15     if (ngx_shm_alloc(&shm) != NGX_OK) {
16         return NGX_ERROR;
17     }
18     // ...
19 }
20
21 // os/unix/ngx_shmem.c:14
22 // shm's points-to set is tainted
23 ngx_int_t
24 ngx_shm_alloc(ngx_shm_t *shm)
25 {
26     // shm->size is tainted
27     shm->addr = (u_char *) mmap(NULL, shm->size,
28                                 PROT_READ|PROT_WRITE,
29                                 MAP_ANON|MAP_SHARED, -1, 0);
30
31     if (shm->addr == MAP_FAILED) {
```

```
32          ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
33                        "mmap(MAP_ANON|MAP_SHARED, %uz) failed", shm->size);
34          return NGX_ERROR;
35      }
36
37      return NGX_OK;
38  }
```

Listing 5.2: Source code for a false positive mmap in `nginx`. The analysis can not distinguish between `shm.log` and `shm.size` because it is not field-sensitive, so it claims that `shm.size` is tainted by `cycle` on line 13. This taint is propagated to the length argument of mmap.

5.4.2 Wireshark

3) Unexploitable exec in `wireshark`. Listing 5.3 shows relevant source code for an unexploitable `exec` in `wireshark`. The tool correctly identifies this; it produces no data-flow path because no such path exists. The program name (`dumpcap`) is appended to the relevant directory path and passed to exec; no data-flow from an indirect call target is possible.

```
1   // capture/capture_sync.c:205
2   /* a new capture run: start a new dumpcap task and hand over parameters
        through command line */
3   gboolean
4   sync_pipe_start(capture_options *capture_opts, GPtrArray *capture_comments,
5                   capture_session *cap_session, info_data_t* cap_data,
6                   void (*update_cb)(void))
7   {
8       // ...
9       char **argv;
10      // ...
11      // argv[0] = "<...>/dumpcap"
12      argv = init_pipe_args(&argc);
13      // ...
14      if ((cap_session->fork_child = fork()) == 0) {
15          // ...
16          // args to exec are clean
17          execv(argv[0], argv);
18          // ...
19      }
20      // ...
21  }
22
23  // capture/capture_sync.c:170
24  /* Initialize an argument list and add dumpcap to it. */
25  static char **
26  init_pipe_args(int *argc) {
27      char **argv;
28      // ...
```

```
29     // target program (dumpcap) is hard-coded
30     exename = g_strdup_printf("%s/dumpcap", progfile_dir);
31     // ...
32     /* Make that the first argument in the argument list (argv[0]). */
33     argv = sync_pipe_add_arg(argv, argc, exename);
34     // ...
35     return argv;
36 }
```

Listing 5.3: Source code for an unexploitable exec in `wireshark`. The arguments passed to exec (`argv` on line 17) are built by calling `init_pipe_args` (line 12), which appends `"/dumpcap"` to a directory path (line 30) and assigns the result to `argv[0]` (line 33).

5.4.3 OpenVPN3

4) False positive write in `openvpn3`. Listing 5.4 shows relevant source code for a write in `openvpn3` which is correctly tainted but ultimately benign. The tainted data-flow path is highlighted in yellow. The source function is `openvpn::OpenSSLContext::Config::new_factory`: It is a legal indirect target because it is a virtual function and therefore address-taken (and also shares a type signature with at least one ICS). All parameters of the source function are tainted, including the implicit `this`, which gets passed to the `openvpn::OpenSSLContext` constructor to become `config_arg`. Eventually, `tls_cipher_list` is loaded out of it and written to a `std::stringstream`. This is the tainted write operation; the written value is the tainted argument. Even though this code does not write to the filesystem, it writes to a string through `std::ostream::operator<<`. As discussed in §5.3, we have to consider all call sites of this function a write sink because it is possible to write to a file through it.

```
1  // openvpn3-core/openvpn/openssl/ssl/sslctx.hpp:101
2  namespace openvpn {
3
4    // Represents an SSL configuration that can be used
5    // to instantiate actual SSL sessions.
6    class OpenSSLContext : public SSLFactoryAPI
7    {
8      // ...
9      class Config : public SSLConfigAPI
10     {
11       // ...
12       // legal indirect target; this is tainted
13       SSLFactoryAPI::Ptr new_factory() override
14       {
15         return SSLFactoryAPI::Ptr(new OpenSSLContext(this));
16       }
17       // ...
18     }
19     // ...
20     // config_arg is tainted, and taints config
```

```
21      OpenSSLContext(Config* config_arg)
22        : config(config_arg)
23      {
24                  // ...eventually...
25                  translated_cipherlist = translate_cipher_list(config-
            >tls_cipher_list);
26
27  // openvpn3-core/openvpn/openssl/ssl/sslctx.hpp:1070
28      // cipherlist is tainted
29      static std::string translate_cipher_list(std::string cipherlist)
30      {
31        std::stringstream cipher_list_ss(cipherlist);
32        std::string ciphersuite;
33
34        std::stringstream result;
35
36        // ...
37        while(std::getline(cipher_list_ss, ciphersuite, ':'))
38          {
39              // ...
40              // tainted value is written to a stringstream
41                result << ciphersuite;
42              // ...
43          }
44        // ...
45      }
```

Listing 5.4: Source code for a false positive write in `openvpn3`. The `this` pointer is tainted in `openvpn::OpenSSLContext::Config::new_factory` (line 13), which is propagated through `config_arg` (line 21) into `translated_cipherlist` (line 25). This value is then written to a `std::stringstream` (line 41), which is a write sink.

5.5 Analysis Cost

To evaluate the scalability of our analysis and the effectiveness of the CFG pruning technique described in §4.2 (EQ4), we measured resource consumption over the test suite. Table 5.5 shows the time and peak memory used to analyze each program, with and without pruning, as measured by `/usr/bin/time`. Taint propagation and CFG pruning were performed in *backward* mode, with the CFI policy set to *addr-type*. While we ran our experiments on a machine with 512GB of memory (§5.1), time became the bottleneck once workloads began to use more than 30GB.

By applying CFG pruning, we see order-of-magnitude reductions in time and memory use in some programs (`firefox`, `nginx`, `xpdf`) and modest reductions in others (`updater`, `dbmail-imapd`, `libdbmail.so`). Despite reducing the memory use in `libwireshark.so`, pruning does not allow the analysis to complete within a reasonable length of time (2 hours). The time reduction seems to be more significant for larger programs, which could be explained by the points-to analysis taking up a larger portion of the total run time due to its quadratic scaling. Since CFG pruning filters the

Program	**Size** (MB)	**Pruning On** **Time** (s)	**Pruning On** **Mem** (GB)	**Pruning Off** **Time** (s)	**Pruning Off** **Mem** (GB)
`firefox`	10.71	7.94	0.91	366.57	1.89
`updater`	0.83	0.50	0.10	0.53	0.12
`crashreporter`	4.24	6.08	0.62	11.27	0.94
`httpd`	3.12	2.42	0.36	3.55	0.51
`libapr-1.so`	1.32	0.85	0.16	2.40	0.29
`nginx`	6.46	10.07	0.88	186.92	4.28
`dbmail-imapd`	0.52	0.32	0.08	0.39	0.11
`libdbmail.so`	1.72	1.82	0.30	2.10	0.35
`psql`	1.20	3.83	0.48	4.48	0.54
`leveldbutil`	0.44	0.59	0.12	1.21	0.19
`wireshark`	75.64	65.25	6.67	288.15	15.05
`tshark`	1.92	1.00	0.26	1.22	0.31
`libwireshark.so`	334.70	–	*37.17	–	*50.69
`libwiretap.so`	4.14	2.82	0.41	4.59	0.59
`openvpn3`	5.54	11.25	1.44	–	*3.27
`openvpn3-admin`	0.90	1.77	0.27	3.42	0.42
`xpdf`	4.34	46.99	3.62	1,123.68	21.49

Table 5.5: Analysis time and peak memory usage over the test suite, with and without pruning. *Size* indicates the size of the bitcode file. *Time* and *Mem* indicate the time and memory usage reported by `/usr/bin/time`. There were 3 failed runs: `libwireshark.so` timed out after 2 hours with and with pruning, and SVF threw an error during SVFG construction on `openvpn3` when run without pruning. For these failed runs, we report the peak memory usage up to the point when the analysis failed or was killed.

set of pointers to be analyzed, it mainly reduces points-to analysis time and therefore has a greater effect when this time is a signficant fraction of the total.

With backward pruning, the set of pointers to be analyzed is determined by the set of basic blocks that can reach a sink through direct control flow. While we expect the number of analyzed pointers to increase with the number of sinks, there are too many confounding factors for us to relate it directly to total analysis time. We leave further exploration of this relationship to future work.

Chapter 6

Limitations and Future Work

Moving forward with syscall reachability analysis, we see several challenges and opportunities.

Improving analysis precision. The set of tainted sinks produced by our analysis represents an upper bound on the exploitability of the program. While it identifies sinks that are actually exploitable (§5.4.1), it also produces false positives (§5.4.1, §5.4.3) for a variety of reasons (§3.5). Some of these false positives can be eliminated by using a more precise pointer analysis (e.g. the one on `nginx` could be eliminated with a field-sensitive analysis), which we consider to be orthogonal to our work. Others require changes to taint propagation: Ensuring that the height of the call stack along the tainted path is never negative could eliminate some false paths with illegal control-flow. False negatives are also a concern, but harder to evaluate. A potential source of error is that our list of libc syscall wrappers was created manually and may be incomplete. This could be addressed in the future by analyzing all functions containing a syscall instruction in an implementation like glibc.

Application to exploit generation. The source-sink paths output by our analysis tool represent data-flow that may be exploitable by an attacker. While we make conservative assumptions (leading mainly to false positives) with the intention of informing possible defences, it could nonetheless be applied to exploit generation to find syscall gadgets that are actually viable. A validation mechanism could be used to reduce the false positive rate: For instance, symbolic execution along the source-sink path could determine the attacker's ability to set a syscall argument to a particular value.

Targeted defences. Our evaluation (§5.3) suggests that lightweight input sanitization can mitigate the harm caused by an attacker in most of the cases examined. In addition to this, the analysis allows the defence to be tailored to the program: Code around tainted source-sink paths could be restructured, or additional domain-specific checks could be added. CFI precision could also be reduced in insensitive areas: The defence could use call-site-insensitive CFI when the ICS has no targets (sources) that taint a sink; call-site-sensitive CFI could be used for the other ICSs. Under our threat model, this would reduce performance overhead without compromising security.

Chapter 7

Conclusion

Control-flow integrity (CFI) has been the subject of intense research and seen significant industry adoption as a defence against control-flow hijacking. While attacks have demonstrated its limitations in fully preventing control-flow hijacking, we argue that CFI can still make end-to-end exploits difficult by reducing an attacker's ability to compromise the system through a *syscall gadget*. In order to evaluate the exploitability of programs under a variety of CFI policies, we design and implement a specialization of static taint analysis called *syscall reachability analysis* to over-approximate the set of syscall gadgets, which we propose as a quantitative upper bound on exploitability. To do this, we assume a strong attacker who has already performed arbitrary computation in userspace but must follow the CFI policy, and find sensitive syscall call sites whose arguments are reachable by dataflow from attacker-controlled values.

We evaluate our analysis over a suite of representative C/C++ programs. We find that most programs do not make very many sensitive syscalls and that in most of the remaining cases examined, the harm can be mitigated through sanitization of syscall arguments. While more precise CFI policies allow fewer syscall gadgets, they do not eliminate them completely. Our analysis allows exploitability to be measured on a program-by-program basis, allowing CFI and other defences to be tailored for higher security or lower overhead as needed. Potential directions for future work include improving the precision of our analysis, reducing the overhead of CFI enforcement without increasing exploitability, and applying the analysis to exploit generation.

Bibliography

[1] *2021 CWE Top 25 Most Dangerous Software Weaknesses*. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[2] Martín Abadi et al. "Control-Flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165.

[3] Steven Arzt et al. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, 259–269. ISBN: 9781450327848. DOI: 10.1145/2594291.2594299.

[4] *BPF: the universal in-kernel virtual machine*. https://lwn.net/Articles/599755/.

[5] Nathan Burow et al. "Control-Flow Integrity: Precision, Security, and Performance". In: *ACM Comput. Surv.* 50.1 (2017). ISSN: 0360-0300. DOI: 10.1145/3054924.

[6] Nicholas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[7] Stephen Checkoway et al. "Return-Oriented Programming without Returns". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, 559–572. ISBN: 9781450302456. DOI: 10.1145/1866307.1866370.

[8] Yueqiang Cheng et al. "ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks". In: *Proceedings 2014 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23156. URL: https://www.ndss-symposium.org/ndss2014/programme/ropecker-generic-and-practical-approach-defending-against-rop-attacks/.

[9] *Control Flow Integrity*. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[10] *Control-flow integrity for the kernel*. https://lwn.net/Articles/810077/.

[11] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. San Diego, California, USA: Association for Computing Machinery, 2007, 482–493. ISBN: 9781595937063. DOI: 10.1145/1250662.1250722.

[12] Lucas Davi et al. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 401–416. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[13] Isaac Evans et al. "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, 901–913. ISBN: 9781450338325. DOI: 10.1145/2810103.2813646.

[14] Xinyang Ge, Weidong Cui, and Trent Jaeger. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace". In: *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017. URL: https://www.microsoft.com/en-us/research/publication/griffin-guarding-control-flows-using-intel-processor-trace/.

[15] *Hardening ELF binaries using Relocation Read-Only (RELRO)*. https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

[16] Hong Hu et al. "Enforcing Unique Code Target Property for Control-Flow Integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, 1470–1486. ISBN: 9781450356930. DOI: 10.1145/3243734.3243797.

[17] Mustakimur Rahman Khandaker et al. "Origin-sensitive Control Flow Integrity". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 195–211. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/khandaker.

[18] Volodymyr Kuznetsov et al. "Code-Pointer Integrity". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[19] *Pointer Authentication on ARMv8.3A*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[20] *ROPgadget - Gadgets finder and auto-roper*. http://www.shell-storm.org/project/ROPgadget/.

[21] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 745–762. DOI: 10.1109/SP.2015.51.

[22] *Security*. https://webassembly.org/docs/security/.

[23] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313.

[24] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337175.

[25] *SLOCCount*. https://dwheeler.com/sloccount/.

[26] Manu Sridharan and Stephen J. Fink. "The Complexity of Andersen's Analysis in Practice". In: *Static Analysis*. Ed. by Jens Palsberg and Zhendong Su. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 205–221. ISBN: 978-3-642-03237-0.

[27] G. Edward Suh et al. "Secure Program Execution via Dynamic Information Flow Tracking". In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. Boston, MA, USA: Association for Computing Machinery, 2004, 85–96. ISBN: 1581138040. DOI: 10.1145/1024393.1024404.

[28] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-Flow Analysis in LLVM". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, 265–266. ISBN: 9781450342414. DOI: 10.1145/2892208.2892235.

[29] Yulei Sui and Jingling Xue. "Value-Flow-Based Demand-Driven Pointer Analysis for C and C++". In: *IEEE Transactions on Software Engineering* 46.8 (2020), pp. 812–835. DOI: 10.1109/TSE.2018.2869336.

[30] *Whole Program LLVM: wllvm ported to go*. https://github.com/SRI-CSL/gllvm.

[31] Jonathan Woodruff et al. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, 457–468. ISBN: 9781479943944.