Abstract of "Improving Application Security at Scale by Reducing System Call and Library Overprivilege" by Nicholas DeMarinis, Ph.D., Brown University, October 2021.

Software is becoming increasingly complex. To keep up with evolving applications, modern operating systems (OSes) provide a rich and continually-growing set of features through OS interfaces, system libraries, and APIs. However, applications typically have indiscriminate access to this entire range of features, regardless of how much functionality is actually required. As a result, programs are often overprivileged, allowing a potential attacker to (ab)use functionality otherwise irrelevant to the program. Existing methods to reduce application privileges are often fragile or have a prohibitive deployment cost, such as source code, which is not always available, or collection of execution traces, which inherently have limited accuracy due to incomplete code coverage.

In this thesis, we present a set of techniques for reducing overprivilege without requiring access to source code or dynamic tracing, thus enabling more widespread deployment. We characterize overprivilege in applications written in type- and memory-unsafe code (C, C++, assembly) and develop low-cost tools to safely and automatically enforce the principle of least privilege by restricting program code to necessary features only. Specifically, we present `sysfilter`, a framework to reduce the attack surface of the OS kernel by reducing overprivilege with respect to the system call API. `sysfilter` builds on contemporary binary analysis to statically identify a program's system call usage by reconstructing its function call graph (FCG) so that unused system calls can be restricted. We use sysfilter to analyze $\approx$30K binaries in a major Linux distribution to 1) characterize system call overprivilege in situ, 2) demonstrate how our techniques can analyze and improve the security posture of a wide range of programs, 3) investigate challenges to improving the precision of our analysis in real-world programs involving dynamically-loaded code and varied privileges across execution phases. Finally, we extend our FCG extraction techniques into a generic framework, `libfcg`, to facilitate new tools to identify overprivilege in other security domains. Our tools and distribution-wide studies demonstrate the security benefits of reducing overprivilege, as well as the feasibility of using static analysis to improve system security in a precise, effective, and scalable manner.

Improving Application Security at Scale by Reducing System Call and Library Overprivilege

by

Nicholas DeMarinis

M.Sc. Brown University, 2019

M.S. Worcester Polytechnic Institute, 2015

B.S. Worcester Polytechnic Institute, 2013

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Computer Science at Brown University

Providence, Rhode Island

October 2021

Thank you to everyone who has supported me on this journey.

You have made me a better person.

This dissertation by Nicholas DeMarinis is accepted in its present form
by the Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date ____________ ____

Vasileios P. Kemerlis, Advisor

Recommended to the Graduate Council

Date ____________ ____

Rodrigo Fonseca, Reader

Date ____________ ____

Michalis Polychronakis, Reader

Approved by the Graduate Council

Date ____________ ____

Andrew G. Campbell, Dean of the Graduate School

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Defining overprivilege

Software frameworks, applications, and interfaces are becoming increasingly complex. In what has been called the "first law of software development" [1], programs have a tendency to increase in size and scope over time. As program requirements evolve, operating system (OS) features, system libraries, and other APIs inevitably add functionality to support new applications and use cases, while often retaining support for older features. This inevitable "feature creep" has serious implications for both application and system security, as programs often have indiscriminate access to all of the functionality provided by OS services, regardless of how much functionality is required. In the event a program is compromised, OS services and libraries can therefore fall victim to the *confused deputy problem* [2], in which an attacker that seizes control of a vulnerable program may ab(use) additional functionality not relevant to the program. We define these at-risk programs as *overprivileged*, as this access to unnecessary system features violates the principle of least privilege [3], effectively increasing the capabilities of an adversary post-compromise.

Overprivilege can take multiple forms, including at the most fundamental methods by which programs interact with the operating system. In particular, the system call (syscall) API [4] provides a range of fundamental OS services—such as allocating memory, executing programs, creating network connections, and performing file system operations—with a total of 352 syscalls available in the latest stable Linux version (v5.8, x86_64). While not all applications use all system calls, each program has *complete* access to the *entire* system call set. For example, in a study of ≈30K applications in Debian `sid` (§ 4.3), only 55% use `execve` to execute new programs, while 34% use `listen` to wait for new network connections. The remainder of these programs–*i.e.*, over half of those observed—are overprivileged with respect to these system calls: they have the capability to run new programs or listen for network connections, respectively, even though they do not use these features. This is a clear violation of the principle of least privilege, which threatens the integrity of the entire system: after gaining control of a vulnerable program, an attacker can utilize additional syscalls not part of the application, or potentially escalate their privileges *even further* by exploiting additional vulnerabilities in less-frequently-used—and therefore less-stressed—kernel code [5, 6, 7].

Overprivilege is also evident in usage of system shared libraries. Consider a simple "hello world" program built for a Linux system that links with a standard C library, e.g., `glibc`. When compiled with `gcc` version 10.2.0 on the latest Debian Linux distribution, the main program comprises less than 200 bytes of code—but it links with `glibc` version 2.31, which contains ≈1.4MB of code [8]. Despite only using the code required for the `printf` function, the program still loads all of the code for `glibc` into its memory. Modern exploits are predominantly built using *code reuse* techniques like return/jump/call-oriented programming (ROP/JOP/COP), where an attacker redirects a vulnerable program's control flow to a series of useful code fragments from the program's address space. Thus, unused library code, or *code bloat* [1] can therefore 1) increase the amount of "ammunition" available to an attacker for code reuse attacks, and 2) increase overhead for software-based defenses like CFI and continuous code randomization, making deployment of these techniques more challenging.

1.2. The need for tools to reduce overprivilege

A number of efforts in both research and industry have explored methods to restrict application privileges to OS services. With regard to the Linux syscall API, there have been attempts to restrict the set of syscalls based on per-application policies. Early examples attempt to model the set of allowed syscalls using automata-based static analysis [9, 10], statistical models similar to intrusion detection [11, 12], compile-time static analysis [13], and dynamic tracing [14]. More recently, high-profile applications like Firefox and OpenSSH, as well as container runtimes, use manually-crafted policies to sandbox their execution using `seccomp-BPF`, aided by dynamic tracing [15]. With regard to addressing code bloat, there have been a number of recent efforts to *debloat* software using a variety of methods to identify unused code sections, including piece-wise compilation [16], user-defined policies and configuration files [17, 18], delta-debugging [19, 20, 21], and dynamic tracing [22].

While these efforts have demonstrated the security benefits of reducing overprivilege in certain contexts, they are often fragile and have a prohibitive cost for widespread deployment. Approaches requiring source code analysis or program recompilation [13, 16, 18] are restricted to applications where source code is available, and require the source code to be present at analysis time. Policies generated using runtime tracing [14, 15, 22] or delta debugging [19, 20, 21] are inherently incomplete for non-trivial programs as they rely on the presence of test cases or runtime traces to provide information about how code is executed. These dynamic analyses require significant effort to develop or generate test cases or traces with sufficiently high coverage for a certain workload, which is itself a significant area of ongoing research. In terms of reducing overprivilege, these limitations are exemplified by the scope of prior evaluations: to the best of our knowledge, all prior works in this area are only tested on a small number (<200) of programs. This demonstrates that existing approaches have a high setup cost per application, due to the overhead associated with recompilation, dynamic testing methods, or creation of user-defined configurations.

Accordingly, there is a need for low-cost methods to automatically reduce overprivilege *at scale*—*i.e.*, for a wide variety of programs, and without the intensive developer oversight required by source code or dynamic tracing. Approaches based on static analysis provide a compelling method to remove the runtime tracing requirement, and current state-of-the-art disassembly tools (§ 3.2.2) show promise for analyzing program control flow without source code. However, there are significant challenges to building robust tools that can a) analyze a wide variety of complex programs, and b) demonstrate effective reductions in application privileges. It is therefore an open question as to the extent which static, binary-only analysis can produce robust and effective tools for reducing overprivilege.

1.3. Thesis goals and contributions

The goal of this thesis is to investigate the extent to which static, binary-only analysis can serve as a low-cost, effective method for reducing overprivilege in type- and memory-unsafe programs. We explore techniques for reducing overprivilege by developing an effective method for identifying a program's required functionality *without* requiring access to source code or dynamic tracing. In doing so, we develop a set of tools that can 1) safely and automatically reduce overprivilege in a wide range of programs and 2) can be used to significantly improve their security posture. The contributions of this thesis are as follows:

`sysfilter` `sysfilter` is a framework to reduce overprivilege with respect to the system call API in order to reduce the attack surface of the OS kernel. We present a set of techniques (Chapter 3) based on contemporary binary analysis tools to identify the set of system calls made by an application binary and its shared libraries by constructing a safe, right overapproximation of the program's function call graph (FCG). Once the system call set has been identified, `sysfilter` restricts the program from executing other syscalls at runtime, thereby enforcing the principle of least privilege with respect to the system call set. In addition to an extensive validation and performance analysis, we perform a large-scale analysis (Chapter 4) to investigate challenges in building our analysis into a robust and scalable tool. We used `sysfilter` to analyze ≈30K C/C++ applications in the Debian `sid` Linux distribution to demonstrate system call overprivilege "in the wild" as well as how our tools can effectively improve the security posture of a large set of programs and systems by restricting syscall sets.

Improvements to callgraph construction and partitioning In Chapter 5, we further explore how our static analysis techniques can be used to limit overprivileging *within* a single program by identifying fixed *partitions* in the program's FCG that can use different enforcement policies. Informed by our large-scale studies on Debian `sid`, we demonstrate areas where our analysis tools can identify partitions based on well-known control flow transition points (such as thread creation) and report on their effectiveness of reducing system call sets at these transition points. In addition, we explore challenges in deploying our analysis tools at scale by characterizing the effect of dynamically-loaded code (*i.e.,* using `dlopen/dlsym`) on system call policies and demonstrate how common-case usages of dynamic loading can be handled in practice.

Generic framework and `libfilter` Motivated by our work with `sysfilter` and other research efforts in software debloating, we extend our FCG extraction techniques as a generic framework, `libfcg` (Chapter 6), that can be used to identify overprivilege in other domains. We use our framework to implement `libfilter`, a software debloating tool to remove unused code from shared libriares. Given a set of application binaries and shared libraries, `libfilter` produces a set of "thinned" libraries that contain only necessary code to support the given applications. We conduct further distribution-wide studies using `libfilter` to demonstrate code bloat and demonstrate that our framework can extract control flow information from a wide range of programs and improve their security posture.

Thesis Statement Modern software requires a rich set of features that are typically provided indiscriminately by the operating system through various interfaces, system libraries, or runtime environments, effectively resulting in overprivileged code. In this thesis, we characterize overprivilege in applications written in type- and memory-unsafe code (e.g., C, C++, assembly) and present a set of techniques to safely and automatically enforce the principle of least privilege by restricting program code to necessary system features only. We demonstrate a set of low-cost tools for identifying overprivilege and restricting access to system call and library APIs, as well as a framework to extend our techniques to other security domains. In doing so, *we improve the security posture of systems by restricting programs to necessary features only, thereby enforcing the principle of least privilege in a precise, effective, and scalable manner*.

Chapter 2

Characterizing overprivilege

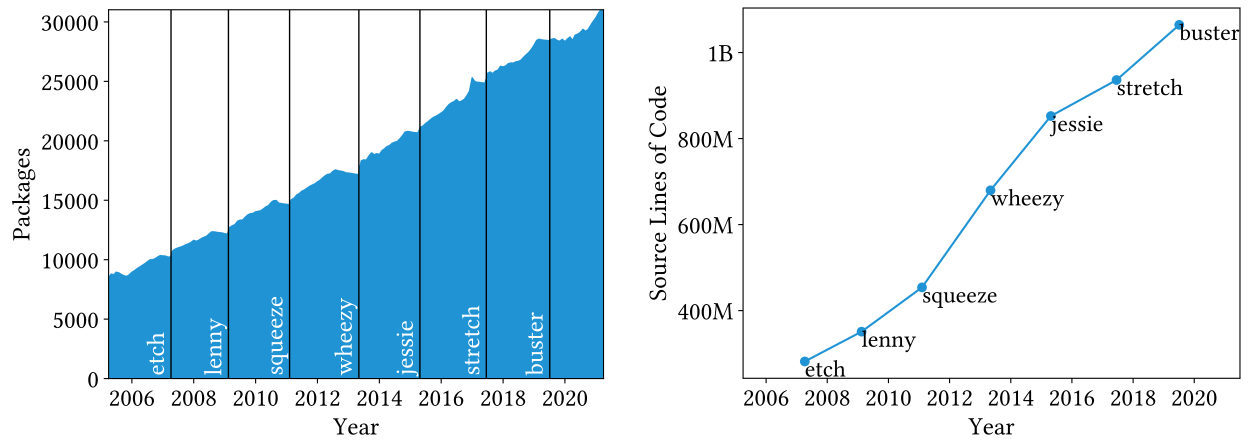
2.1. What is overprivilege?

Overprivilege has resulted from the inherent complexity in modern software. As applications, libraries, and interfaces have evolved, there is an inexorable tendency for software to increase in both size and complexity. Holzmann [1], cites this trend as the "first law of software development," stating that software tends to grow over time to keep up with modern standards and requirements.

Figure 2.1 demonstrates this trend for the Debian Linux distribution. Over the past 10 years, the number of software packages (*i.e.,* installable software units) in Debian's `stable` version has nearly doubled (Figure 2.1)—in 2021, the repository now has over 30K packages that must be maintained by distribution developers. The *amount* of code to support these applications has similarly increased (Figure 2.1b), from approximately 400 million source lines of code in the "squeeze" release in 2011, to over 1 *billion* lines of code the current version, "buster," in 2021, a 2.3x increase.

As programs evolve, it demands new functionality of the operating system (OS) APIs and system libraries that support them. As these system services add new functionality to support new applications and use cases, they often retain support for older features, thus rendering them subject to an inevitable *feature creep* where more and more code is accumulated to support a wide range of programs. Feature creep in system libraries and APIs can have serious implications for security, since programs often have indiscriminate access to *all* functionality provided by these OS services, regardless of how much functionality the program actually requires for operation.

Since OS services inherently provide key system functionality (e.g., accessing the network, performing filesystem operations, manipulating permissions), access to each feature represents a



(a) Total number of packages **(b)** Source lines of code (SLOC)

Figure 2.1: Size of Debian `stable` repository over time. We show the number of packages (a) and total lines of code (b) contained in Debian's `stable` distribution throughout its release history, as reported by Debian's repository monitoring tools [23].

critical security resource. If an attacker gains control of a vulnerable program using exploitation techniques such as code injection or code reuse, they can ab(use) any OS features to which the program has access. Thus, access to additional, unnecessary OS features increases the capabilities of an attacker post-compromise while providing no benefit to the original program. We define these programs as *overprivileged*, as this access to additional features violates the principle of least privilege [3], exacerbating the risk to the system as a whole by increasing the capabilities of an adversary after a program is compromised.

In the remainder of this chapter, we discuss two sources of overprivilege that are pervasive in modern operating systems, syscall API usage, and library code bloat, and the threats they pose to system security. We review existing research efforts to reduce applications privileges in these areas, highlighting challenges to deploying these methods at scale. In doing so, we demonstrate the need for development of low-cost tools to identify and reduce overprivilege in a precise, scalable, and effective manner in order to improve security system-wide.

2.1.1. Overprivilege in system calls

The system call (syscall) API [24] provides the core interface for applications to access fundamental OS services, such as allocating memory, executing programs, creating network connections, and more. On the x86-64 architecture, system calls are performed using the `syscall` instruction, where the value in the `rax` register holds the *syscall number*, an integer that selects the OS feature being requested. Some system calls are virtually ubiquitous in all programs, such as

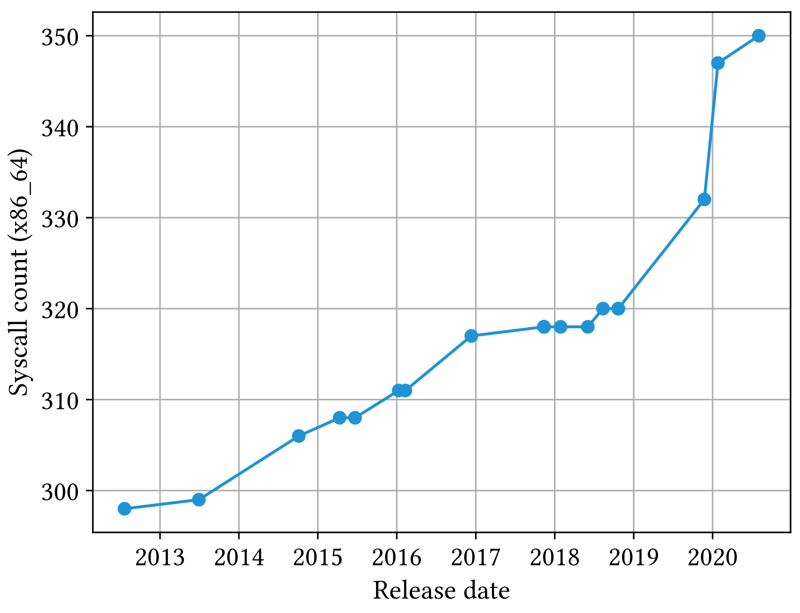


Figure 2.2: Size of system call API in x86_64 Linux on a single system. Number of syscalls gathered from kernels used on a Gentoo Linux system. Syscall set size was determined based on the number of x86_64 syscalls listed in the file `arch/include/generated/asm/syscalls_-64.h`. This is an empirical measurement from a single system and does not reflect the growth between periodic kernel releases. Nevertheless, it demonstrates significant growth in the syscall API over time, with >50 syscalls added since 2013.

`read` and `write`, while others provide access to fundamental, and, often, security-critical, features, such as creating or opening network connections (e.g., `connect`, `socket`, `bind`), creating threads or processes (`execve`, `clone`), or managing permissions (`chmod`).

On Linux, the size of the system call API has grown throughout its lifetime. In a study of the kernel source from 2005–2014, 76 system calls were added to the API, but only 6 were removed. Figure 2.2 shows the growth of the syscall API over time since 2013. At the time of this writing, the current stable version of Linux (v5.8) provides a total of 352 for the x86-64 architecture. System calls are typically added to accommodate new kernel features, or to expand the capabilities of existing interfaces. As an example of the former, kernel v5.1 (released in 2019) added 4 new system calls to support a new asynchronous I/O interface `io_uring` [25]. In the latter case, some system calls duplicate existing functionality, but add new capabilities: for example, the kernel has *three* versions of the "open" syscall to create file descriptors: `open`, `openat`, and `openat2`—the latter was added in v5.6 (2020) to provide additional error handling.

Accordingly, the size of the system call API is a significant source of overprivilege. While not all applications use all system calls, each application has full access to the *entire* syscall set. Thus, an attacker that gains control of a vulnerable program may make use of syscalls unused by an application, demonstrating a clear violation of the principle of least privilege. For example, a simple shell utility like `cat` or `grep` may not need to access the network, but if an adversary can exploit a vulnerability such a program and achieve arbitrary code execution, they could make arbitrary system calls and, e.g.,, listen for network connections by opening a socket. More critically, access to the entire syscall API also increases the attack surface of the operating system itself. An attacker already in control of a program could potentially exploit additional vulnerabilities in less-used—and therefore potentially less-stressed—system calls, thus using a kernel vulnerability to escalate their privileges *even further* [5, 7, 26, 27, 28, 29]. As one example, the `ptrace` system call, used for runtime process tracing, was responsible for a privilege escalation vulnerability (CVE-2019-13272 [30]) for kernels v5.1 and earlier. Even though `ptrace` is used by very few programs directly (§ 4.3.4), *all* applications have the ability to use this system call by default and thus are capable of triggering the vulnerability in an unpatched system.

2.1.2. Overprivilege from code bloat

Shared library code represents another source of overprivilege. Shared libraries inherently contain code for use by many programs, but not all code a single library may be used by all programs that require it. On Linux, most C/C++ applications link with, at minimum, a standard C library to provide required functionality for program startup as well as helper utilities for common OS and system features. On the latest Debian Linux distribution, a simple "hello world" program compiled with `gcc` has less than 200 bytes of code in its `.text` section (*i.e.,* located in the program's own binary), yet the program requires `glibc` version 2.31, which contains ≈1.4MB of executable code. Even though the program may only require a small fraction of `glibc` code (*i.e.,* for program startup and the `printf` function), all of the code for `glibc` (and any other required shared libraries) is loaded into the program's address space.

This extra code, or `code bloat` can constitute a form of overprivilege as it can provide additional resources for an attacker after a program is compromised. Modern exploits constructed using code reuse techniques such as return/jump/call-oriented programming (e.g., ROP, JOP, COP, *etc.*) [31, 32, 33, 34, 35, 36, 37], in which an attacker redirects a program's control flow to a series of useful code fragments, or `gadgets`, in the program's address space. Accordingly, unused library code can increase the amount of "ammunition" available to an attacker to carry out an attack using code reuse. Further, code bloat can increase overhead for many software-based defenses against code reuse and other control-flow hijacking attacks, such as Control Flow Integrity (CFI) [38, 39, 40] and continuous code randomization [41], which rely on instrumenting and runtime rewriting, respectively, to restrict the program's control flow, thus increasing the deployment cost of these defenses.

2.2. Efforts to reduce application privileges

In this section, we provide an overview of existing research efforts to reduce application privileges in terms of system call API usage and library code usage. We present these works through the lens of identifying effective, scalable methods that can support analysis on a wide range of programs. To assess scalability, we focus on enumerating the tradeoffs in requirements for each type of analysis, such as the types of program information required and the amount of per-program intervention required by a developer, in addition to understanding their overall effectiveness for reducing privileges.

Accordingly, for many areas we distinguish between approaches using static and dynamic analysis methods, as this classification has a significant impact on scalability. In broad terms, static program analysis methods infer properties about a program solely by examining its program code, or a representation thereof, *before* it is executed. These methods require some representation of the program, such as the source code, a compiled binary, or some intermediate representation thereof, which is parsed to determine certain properties about how the program would act during execution. In contrast, dynamic methods identify program features *at runtime* by tracing or otherwise measuring program behavior based on some input. In general, dynamic analysis methods require test cases or traces of valid program behavior to exercise the program features being analyzed. For a non-trivial program, this process is inherently incomplete as it is generally not feasible to enumerate all possible inputs to the program. Thus, building or generating test cases or traces can require significant effort to obtain sufficiently high coverage to represent a certain workload.

In order to develop an effective, *scalable* approach for reducing application privileges, we focus on using static analysis techniques, as these can support analysis of a wide range of programs without requiring costly developer intervention to build or maintain test cases for each program. In the following sections, we demonstrate the challenges inherent to such an analysis and show that there is a need for low-cost, effective analysis tools to identify and reduce overprivilege at scale.

2.2.1. Reducing system call API usage

Early efforts: host-based intrusion detection System call usage patterns have been extensively studied for host-based intrusion detection since the late 1990s. These works mainly focus on developing models for the program's system call usage that can be checked against real-time traces to detect anomalous behavior. Types of models used for these works have included fixed-length sequences of system calls [42, 43], statistical and other learned models [11, 12, 44, 45], interactive policy generation [14], and automata-based representations [10, 46, 47, 48]. Of these, most approaches rely on using runtime tracing [11, 12, 14, 42, 43, 44, 45], to "train" or build models of normal program behavior to. An notable outlier among these is the work of Wagner and Dean [9], which constructed models of a program's system call usage by extracting a form of a program's control-flow graph using static source code analysis. In terms of static analysis, these works share similar challenges to later works, and our own, on attack surface reduction, but applied to the context of intrusion detection, which has different requirements on soundness and requires relatively complex models of program behavior.

System call filtering More recent efforts have focused on identifying and restrict system call usage, known as *system call filtering*. Rather detect whether a process has been compromised, syscall filtering aims to reduce the capabilities of an attacker by restricting the number of system calls the process can make in the first place. Instead of requiring a potentially complex model of how the program operates, the most common form of syscall filtering approaches only require the set of system calls—and, potentially, their arguments—required by the application. This set of system calls defines a *policy* that can be enforced at runtime, thereby *sandboxing* the process to use only the system calls specified by the policy, reducing an attacker's capabilities after a program is compromised. System call filtering is widely deployed in practice in certain contexts, but mainly using manually-generated policies for specific contexts.

Enforcing system call policies The Linux kernel includes built-in support for system call filtering as part of its seccomp [49] framework. Using the `prctl` or `seccomp` system calls, a process can attach a BPF [50] program that is executed at each syscall invocation, allowing programmable filters.[1] `seccomp-BPF` is currently used by a number of security-critical applications to sandbox certain high-risk components. Firefox [52] and Chrome [53] use `seccomp-BPF` to sandbox

[1]Note that `seccomp-BPF` only supports *classic* BPF programs. Extended BPF (eBPF), which supports persistent state and other, more expressive, features, is not supported. Proposals to add eBPF support to seccomp were considered in February 2018 [51], but appear to have been dropped due to security and performance concerns.

rendering processes, while OpenSSH [54] uses uses it to provide privilege separation between the main `sshd` server process and client-handling child processes. These applications use custom, developer-created syscall filtering policies that must be maintained by their respective developers.

Container runtimes like Docker [55] and Podman [56] install `seccomp-BPF` policies for all containers by default and supports using custom policies on a per-container basis. By default, Docker applies a manually-crafted, one-size-fits-all policy that disables 44 syscalls [57] (out of a total of 352 on Linux kernel v5.8). This default profile aims to prevent containers from making system calls related to privileged operations that should only be performed by the host (e.g., `shutdown`, `load_module`) as well as those that might otherwise allow a container to modify host resources. Users may optionally start containers with a different policy, which has created interest generating more fine-grained, container-specific policies.

Aside from `seccomp-BPF`, Systrace [14] and Ostia [58] are earlier examples of userspace agent processes that can enforce custom system call policies, at the cost of a round-trip to userspace for each system call, plus additional overhead for the verification process. The work of Zeng et al. [59] provides an in-kernel alternative to filtering with `seccomp-BPF` by creating per-process system call tables. This improves performance of the filtering operation by only requiring a table lookup per system call, rather than evaluation of a cBPF program.

Automatic generation of system call filters A line of recent work has focused on identifying system calls automatically, mostly in the context of building `seccomp-BPF` policies for containers. Much of this work has used dynamic analysis build syscall policies by tracing runtime behavior. Podman has support for tracing syscalls dynamically with `ptrace` to build container policies [15]. DockerSlim [60] is a tool to remove unneeded files and generate `seccomp-BPF` policies from Docker images using dynamic tracing and HTTP API probing. Similarly, Wan et al. [61] use dynamic analysis to generate `seccomp-BPF` policies automatically. SPEAKER [62] extends this to separate a container's system calls into two phases: those required to start the container, and those required for normal operation, using time-based heuristics to determine the boundary between profiling phases.

Comparatively, relatively few works have investigated generation of system call using static analysis, for containers or for single programs. To address the inherent completeness issues with dynamic generation of container policies, Confine [63] is a (mostly) static analysis-based system for automatically extracting and enforcing syscall policies. Confine analyzes binaries present in container images, but requires access to the source code of the C library (e.g., `glibc` or `musl`).

Zeng et al. [59] also infer valid syscall sets for generic (*i.e.,* not containerized) binaries from using static analysis, but this approach has several limitations affecting soundness. Ghavamnia et al. [13] uses static analysis of source code to perform *temporal specialization*, to generate system call policies for server applications before and after program startup, based on manual annotations to separate program phases. We discuss these works using static analysis in more detail in § 3.2.2.

2.2.2. Reducing code bloat

Efforts to reduce code bloat have been investigated by research efforts in *softwrae debloating*. Software debloating involves removing or restricting access to code regions not used by an application. Related techniques have been studied to reduce memory usage on embedded systems [64, 65, 66] and improve performance. In a security context, debloating reduces the amount of code available to an attacker once a process has been compromised. For example, reducing the amount of accessible code in a process' memory, software debloating limits ability to harvest gadgets for mounting code reuse attacks [31, 32, 33, 34, 35, 36, 37], potentially reducing their post-exploitation capabilities.

Broadly, this entails identifying code sections of a binary or shared libraries that are used in a target application. Using this information, regions of code that are *not* used by the application can be removed, or otherwise made inaccessible to the program at runtime. Recent approaches have explored various methods of identifying utilized code, including input and configuration specifications [17, 18], dynamic tracing [22] and testing [19, 20, 21, 67] methods to find used code paths based on a series of test inputs, to static analysis-based approaches [16, 68, 69] that build a CFG of the program's execution. Another key variation is the method of restricting unused code: approaches that operate on application binaries (*i.e.,* binary files containing a `main`) often rewrite binaries or modify compilation to remove unneeded sections [67, 67, 69], while approaches that also remove code from shared libraries must either *specialize* libraries to a specific set of binaries, or dynamically adjust what library code is loaded for each binary that uses it [16, 68].

"Piece-wise" loading Early approaches focused on debloating shared libraries by removing unused components from the program's memory at load time. In 2015, Mulliner and Neugschwandtner [68] developed an debloating mechanism for Windows DLLs. Their approach disassembles target applications and their libraries to reconstruct the program's CFG. A runtime component loads the CFG information before application startup and replaces unused portions of DLLs with halt instructions to prevent execution. Mulliner and Neugschwandtner note several issues reconstructing the CFG from static analysis that required manual intervention to prevent certain functions from being removed. Specifically, they utilize bounded address tracking [70] to resolve indirect function calls, which is incomplete and can therefore generate a CFG that is too restrictive. Despite these limitations, their approach demonstrated significant code reduction, stripping 28% of code from Adobe Reader. Subsequent works have aimed to improve CFG generation, but many require source code.

Quach et al. [16] perform software debloating on Linux systems. They develop an LLVM-based *piece-wise* compiler to embed information about the program's CFG into each program at compile time and use a custom, piece-wise loader[2] to load only required portions of shared libraries into memory. The authors were unable to debloat `glibc`, which does not compile in LLVM. When debloating other libraries, or those compiled against `musl`, they demonstrated significant reductions in both code size and number of code reuse gadgets. While these approaches demonstrated significant reductions in code size, using a custom loader to selectively identify code regions to load increases program startup times, which may not be desirable in certain applications.

Feature removal Another line of works focus on debloating by *feature removal*, using a user-defined configuration to identify program features not required for a certain use case and generate *specialized* versions of programs (or their libraries) tailored for the application. Trimmer [18] performs argument-based specialization, using a list of user-specified configuration flags that are fixed at constant values to remove portions of the program depending on other values. Koo et al. [17] perform library debloating based on user configurations for modular applications, such as webservers. Their compilation-based approach maps configuration directives to shared libraries and generates targeted server binaries to exclude features for unused configuration directives. Davidsson et al. [71] use a compilation-based approach to create application specific shared libraries for use with web applications. During the compilation process, they add domain knowledge about the target web application and script runtimes it uses (e.g., PHP) to generate shared libraries limited to only the functions required by the script interpreter for the target application. While this generates highly specific libraries, it shows promise for containerized web applications distributed as a whole system image.

[2]The concept of the piece-wise loader, selectively loading components of shared libraries based on a certain program's usage is similar to Mulliner and Neugschwandter's earlier work, though Quach et al. coined the term—we use the term "piece-wise" broadly to refer to both works.

Category	Work	Analysis	Tracing	Config	Apps Tested
Loader-based	Mulliner et al. [68]	Binary		●	1
	Quach et al. [16]	Source			109
Feature removal	Trimmer [18]	Source		●	13
	Koo et al. [17]	Source		●	3
	Davidsson et al. [71]	Source		●	5
Delta debugging	C-Reduce [19]	Source	●	●	–
	Perses [20]	Source	●	●	20
	CHISEL [21]	Source	●	●	10
Static disassembly	Razor [22]	Binary	●		13
	Nibbler [69]†	Binary			117

Table 2.1: Comparison of software debloating approaches. For each work, we specify if the approach uses source code or binary analysis, runtime tracing/test case evaluation, or requires user or developer-specified configurations or instrumentation. "Apps Tested" indicates the number of programs tested (for correctness) as part of the program's evaluation.
†: Nibbler, which was developed concurrently to our work, shares many design goals with our efforts. We implement Nibbler's analysis using our tools to produce `libfilter`, a debloating tool with improved performance compared to Nibbler's original prototype (§ 6.2).

Delta debugging A line of works has investigated program minimization using delta debugging [72, 73] techniques. Generally, delta debugging involves generation of a minimal program satisfying a property or test case. C-Reduce [19] and Perses [20] use delta-debugging to generate minimized versions of programs based on test cases, aimed at generating example programs demonstrating compiler bugs. CHISEL [21] extends this work with reinforcement learning to speed up example generation for large programs, in the context of debloating for to reduce a program's attack surface. All of these approaches require source code for iterative recompilation of minimized programs depend on test cases to exercise required code.

Debloating without source code Razor [22] aims to perform debloating with a combination of static and dynamic analysis. Razor does not require source code, and instead implements a basic-block level dynamic tracer to reconstruct the program's CFG from a set of traces on test inputs. The authors aim to address some of the incompleteness in dynamic tracing by developing several heuristics using static analysis techniques for incorporating *related code* into minimized programs, in addition to blocks exercised by tracing.

Concurrently and independently with our work, Agadakos et al. developed Nibbler [69], which performs debloating on shared libraries using static binary analysis. We extended Nibbler together to build our library debloating tool, `libfilter`, which addresses some issues in Nibbler's initial prototype to produce a more complete analysis. We discuss Nibbler in further detail in § 6.2.

Otherwise, we summarize relevant efforts in software debloating in Table 2.1, highlighting the differences in required inputs to the analysis that would require per-application such as runtime traces or per-application configurations. These metrics indicate the amount of human input required per-application, a critical factor required for analyzing programs at scale. Only Quach et al.'s piece-wise compilation and Nibbler can be considered fully "automatic" methods, as they do not require runtime tracing or per-application configurations. These works analyzed the largest number of programs, which shows promise for scalability; however, piece-wise compilation operates at compile time, limiting its deployment to applications where source code is available. Overall, this demonstrates the need for static, binary-only analysis tools to reduce code bloat across a wide range of programs.

2.3. The need for scalable tools to reduce overprivilege

In this chapter, we have introduced overprivilege in two forms, system call API access and code bloat, and describe how such violations of the principle of least privilege constitute a security risk. In both we cases, we have shown how there is a need for static binary analysis tools to reduce privileges in an effective and scalable manner. Prior approaches have lacked soundness by relying on dynamic testing or incomplete callgraph analysis methods, which have a high per-application cost that precludes large-scale analysis. We have also demonstrated that reducing application privileges in both domains leverage the same principle: determining a representation of the program's call graph to from which to identify required components. In the next chapter, we present `sysfilter` and our function call graph (FCG) extraction techniques designed to provide a safe over-approximation of the program's FCG to enable construction of more scalable tools reduce overprivilege to improve program and system security.

Chapter 3

`sysfilter`: Automated system call filtering

3.1. Overview

`sysfilter`, performs system call filtering by statically and automatically identifying a program's system call footprint and limiting the system calls a program can make at runtime. `sysfilter` aims at mitigating the effects of application compromise by *restricting access to the syscall API* [4]. This 1)limits post-exploitation capabilities [14], and 2) prevents compromised applications from escalating their privileges further via exploiting vulnerabilities in unused, or less-stressed, kernel interfaces [5, 7, 26, 27, 28, 29].

Applications primarily interact with the kernel via the syscall API in order to perform useful tasks. Indicatively, at the time of writing, the Linux kernel (v5.8) provides support for 352 syscalls in x86-64. However, despite the fact that applications only require access to part of the syscall API to function properly (e.g., non-networked applications do not need access to the `socket`-related syscalls), the OS kernel provides full and unrestricted access to the entirety set of syscalls. This approach violates the *principle of least privilege* [3], enabling attackers to utilize additional OS services after seizing control of vulnerable applications. By restricting access to certain syscalls, `sysfilter` naturally limits what OS services attackers can (ab)use and enforces the principle of least privilege with respect to the syscall API—thus, programs are allowed to issue only developer-intended syscalls. In other words, if the developer of an application is not using, say, `socket`, then even if the application is compromised, the attacker will not be able to issue that syscall and acquire a network socket. Similarly, if the application is not (legitimately) invoking `execve`, then

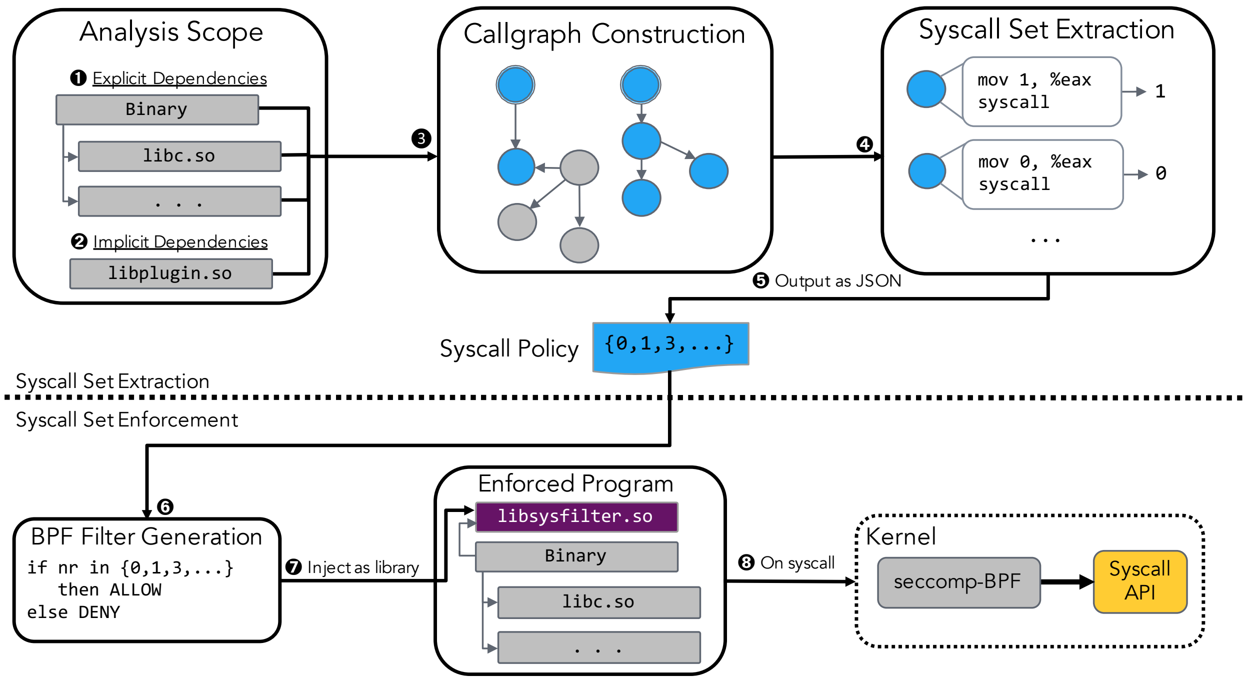


Figure 3.1: `sysfilter` architecture. `sysfilter` consists of two parts: system call set extraction (top) and the system call set enforcement (bottom).

an attacker will not be able to issue that syscall and execute programs.

Further, multiple studies have repeatedly divulged that the exploitation of vulnerabilities in kernel (or in even lower-level, more-privileged [74, 75]) code is an essential part of privilege escalation attacks [5, 7, 26, 27, 28, 29]. To this end, `sysfilter` reduces the *attack surface* of the kernel by restricting the syscall set available to userland processes, effectively providing *defense-in-depth* protection.

`sysfilter` consists of two parts (see Figure 3.1): a syscall-set *extraction* component, and a syscall-set *enforcement* component. The extraction component uses our FCG extraction tools to construct a *safe* over-approximation of the program's FCG across all objects in scope. Finally, it performs a set of program analyses atop the FCG in order to make the over-approximation as tight as possible and construct the syscall set in question. The tasks above are performed *statically* and the syscall set returned by the extraction tool is *complete*: *i.e.*, under any given input, the syscalls performed by the corresponding process are guaranteed to exist in the syscall set—this includes syscalls that originate from the binary itself, `libc`, or any other dynamically-loaded shared library. The latter part enforces the extracted set of syscalls, effectively *sandboxing* the input binary. Specifically, given a set of syscall numbers, the enforcement tool converts them to a `BPF` program [76] to be used with `seccomp-BPF` [77].

3.1.1. Background and threat model

Adversarial capabilities In this work, we consider userland applications that are written in memory-unsafe languages, such as C/C++ and assembly (ASM). The attacker can trigger vulnerabilities, either in the main binaries of the applications or in the various libraries the latter are using, resulting in memory corruption [78]. Note that we do not restrict ourselves to specific kinds of vulnerabilities (e.g., stack- or heap-based memory errors, or, more generally, spatial or temporal memory safety bugs) [79, 80] or exploitation techniques (e.g., code injection, code reuse) [31, 32, 35, 78, 81].

More specifically, the attacker can: (a) trigger memory safety-related vulnerabilities in the target application, multiple times if needed, and construct and utilize exploitation primitives, such as arbitrary memory writes [82] and reads [83]; and (b) use, or combine, such primitives to tamper-with critical data (e.g., function and `vtable` pointers, return addresses) for hijacking the control flow of the target application and achieve arbitrary code execution [84] via means of code injection [81] or code reuse [31, 32, 33, 34, 35, 36, 37]. In terms of adversarial capabilities, our threat model is on par with the state of the art in C/C++/ASM exploitation [78, 85]. Lastly, we assume that the target applications consist of *benign* code: *i.e.,* they do not contain malicious components.

Hardening assumptions The primary focus of this work is modern, x86-64 Linux applications, written in C, C++, or ASM (or any combination thereof), and compiled in a *position-independent* [86] manner via toolchains that (by default) *do not mix code and data* [87, 88], such as GCC and LLVM.[1] In addition to the above, we assume the presence of *stack unwinding information* (`.eh_frame` section) in the ELF [89] files that constitute the target applications.

In § 3.2, we explain in detail the reasons for these two requirements—*i.e.,* position-independent code (`PIC`) and `.eh_frame` sections. However, note that (a) `PIC` is enabled by default in modern Linux distributions [90, 91], while (b) `.eh_frame` sections are present in modern GCC- and LLVM-compiled code [88, 90]. The main reason for (a) is full ASLR (Address Space Layout Randomization): in position-dependent executables ASLR will only randomize the process stack and `mmap`- and `brk`-based heap [92]. Moreover, `PIC` in x86-64 incurs negligible performance overhead due to the existence of `PC`-relative data transfer instructions (`%rip`-relative `mov`) and extra general-purpose registers (16 vs. 8 in x86). As far as (b) is concerned, stack unwinding information is mandated by C++ code for exception handling [93], while both GCC and LLVM emit

[1]Andriesse et al. [87], and Alves-Foss and Song [88], have independently verified that modern versions of both GCC and LLVM *do not* mix code and data. `icc` (Intel C++ Compiler) still embeds data in code [88].

`.eh_frame` sections even for C code to support interoperability with C++ [90] and various features of certain `libc` (C library) implementations—e.g., `backtrace` in `glibc` (GNU C Library).

Lastly, we assume a Linux kernel with support for `seccomp-BPF` (SECure COMPuting with filters) [77]. (All versions $\geq$ v3.5 provide support for `seccomp-BPF` in x86-64.) Every other standard userland hardening feature (e.g., NX, ASLR, stack-smashing protection) is orthogonal to `sysfilter`; our proposed scheme does not require nor preclude any such feature. The same is also true for less-widespread mitigations, like CFI [94, 95], CPI [96, 97], code randomization/diversification [41, 85, 98], and protection against data-only attacks [99].

3.2. `sysfilter` design

3.2.1. Analysis scope

A crucial first step toward constructing a precise callgraph is collecting all code that may be used by the target application. This is a complex processes due to the various nuances in how shared libraries may be loaded. The input to `sysfilter` is an (x86-64) ELF file that corresponds to the main application binary. To aid in determination of function boundaries, `sysfilter` requires `PIC` binaries as input—this is is the default setting in modern Linux distributions [90, 91]. Once `sysfilter` verifies that the main binary is indeed `PIC`, it adds it to the *analysis scope*, and proceeds to resolve dependencies regarding dynamic shared libraries. The process for this step is similar to the dependency resolution of `ldd`. In addition to the above, it is also possible to provide as input a set of *implicit* dynamic shared object dependencies to `sysfilter`: *i.e.*, a list of additional `.so` ELF files that need to be added to the analysis scope, irrespectively of whether they exist in any of the loaded objects' `.dynamic` section. This functionality is important in order to support the analysis of binaries that have run-time dependencies to shared objects (e.g., via `dlopen`) or use `LD_PRELOAD`.

3.2.2. Function-call graph construction

Once every ELF object is added to the analysis scope, `sysfilter` proceeds with the construction of the *function-call graph* (FCG) of the *whole* program. The FCG contains parts of the code (functions) that are reachable, under any possible input to the corresponding process. Critically, not all code used by the libraries in the analysis scope is used is used: e.g., applications that link with `libc`, `libpthread`, `libdl`, *etc.*, do not use all functionality in `libc`—usually, only part of library functionality is utilized [16, 69]. Accurately identifying unreachable code is crucial: identifying unused functions excludes them from our FCG, providing security benefits when code is removed or restricted for debloating. However, incorrectly excluding code is potentially catastrophic, as it can cause application crashes. We describe how our callgraph construction process in two steps: 1) accurately disassembling ELF files to identify the boundaries between functions, and 2) building the callgraph by recovering edges from function calls.

Precise disassembly Obtaining the complete and precise disassembly of arbitrary binary programs is an undecidable task [100]. The problem stems from two main reasons: (a) not being able to decisively differentiate code from data [100]; and (b) not being able to precisely identify function boundaries [101, 102]. Fortunately, modern toolchains, like GCC and LLVM, (1) do not mix code and data [87, 88] and (2) embed stack unwinding information to (x86-64) C/C++ binaries [90].

`sysfilter` takes advantage of (1) and (2) in order to precisely disassemble the executable code from all ELF files in the analysis scope. More specifically, for each `.so` object in the analysis scope, `sysfilter` uses the stack unwinding information (`.eh_frame` section) to get the exact boundaries of all functions in executable sections (e.g., `.text`, `.plt`). Moreover, special care is taken to correctly identify functions of `crtstuff.c` (`libgcc`), which are compiled into the sections `.init` and `.fini`, as well as `crtbegin.o`, and executed as part of the dynamic linker/loader.

Armed with precise information about function boundaries, and given the strict separation of code and data, `sysfilter` performs a *linear sweep* [103] in all code regions that correspond to identified functions, to disassemble their executable code. The resulting disassembly does not contain any invalid instruction, due to data treated as code or incorrect function boundary detection, nor does it miss instructions due to unidentified code—the resulting disassembly is complete, precise, and accurate.

Callgraph construction: Direct call graph Using the precisely-disassembled function information, `sysfilter` proceeds to construct the FCG of the input program. Broadly, this entails recovering edges between the identified functions by following control flow transfer instructions. In the general case, this is a hard problem due to the various types of control flow transfers, the varying amount of information that can be recovered about control flow targets statically.

As a first step, `sysfilter` contructs the *direct call graph* (DCG), which is the part of the FCG that corresponds to directly-invoked functions/code. This is achieved by first adding to the DCG the entry point of the main binary (*i.e.*, the setup functions called before `main`), followed by all the functions whose addresses are stored at the subsequent (ELF) sections: `.preinit_array`, `.init_array`, and `.fini_array`; the code/function in `.init` and `.fini` is also added to the DCG. Subsequently, the same process is repeated for every other `.so` ELF object in the analysis scope. At the end of this step, a set of *initial* functions are added to the DCG, which correspond to the entry points of the code that is executed during the initialization/finalization of the respective process by the dynamic linker/loader (`ld.so`). `sysfilter` also supports adding an additional set of implicit function dependencies specified externally, which this is required to aid the analysis of binaries that have run-time dependencies to certain functions (e.g., via `dlsym`) or use `LD_-PRELOAD` [104].

Next, the code of each such function in the DCG is linearly scanned to identify direct `call` instructions, which instruction encode the target address as a `%rip`-relative immediate operand. Since the value of `%rip` is known during the linear sweep, the target functions of these instructions (callees) can be statically identified, and do not change at run-time.

Branch instructions, like (un)conditional `jmp` instructions, that cross function boundaries, are also considered as they are typically used for implementing *tail-call elimination* [105]. Each identified target function (callee) is also added to the DCG, and the process is repeated until no additional functions can be added. Cross-shared library calls, via the Procedure Linkage Table (PLT) [106], are handled by inspecting the `.dynsym`, and `.dynstr`, sections of the ELF files in scope and "emulating" the binding (symbol resolution) rules of `ld.so` [104, 107]. This is analogous to executing `ld.so` with '`LD_BIND_NOW=1`'.

The net result of this process is a subset of the FCG rooted from the program's contains the entry point(s) and initialization/finalization functions of the ELF objects in scope (plus the implicitly-added functions, if provided), followed by every other function that is *directly-reachable* from direct control flow transfers.

Callgraph construction: Address-taken call graph The direct call graph process does not account for functions targeted by *indirect* `call`/`jmp` instructions. These instructions have as operand a (general-purpose) register, or a memory location, which stores the target address, *i.e.*, the address of the callee. Indirect control flow instructions are typically used for dereferencing function pointers (C/C++) and implementing dynamic dispatch (C++) [108]. Statically resolving the target addresses of indirect `call`/`jmp` instructions is a hard problem [109], mostly due to the imprecision of *points-to* analysis [110]. Indeed, all of the authors of the approaches using static analysis for system call filtering [10, 46, 59] and software debloating [22, 68, 69] discussed in Chapter 2 cited issues resolving indirect control flow transfers as affecting the soundness of their results.

`sysfilter` handles this problem by *over-approximating* the FCG by constructing the *address-taken call graph* (ACG). The ACG is *complete*: *i.e.,* it never excludes functions that can be executed by the program (under any possible input). This concept is similar to the handling of address-taken (AT) functions used by Nibbler [69]. `sysfilter` extends this approach using Egalito's state-of-the-art precise disassembly techniques, including the best known jump table detection and handling to date, improving the precision of the control-flow and dataflow analyses.

The first step to construct the ACG is to identify all *address-taken* functions: *i.e.,* functions whose address appears in `rvalue` expressions, function arguments, `struct`/`union` initializers, and C++ object initializers, or functions that correspond to *virtual* methods (C++). The set of all address-taken (AT) functions is a superset of the possible targets of every indirect call site in scope. This is because indirect `call`/`jmp` instructions take as operands (general-purpose) registers, or memory locations, which can only hold absolute addresses; therefore, in order for a function to be invoked via an indirect `call`/`jmp` instruction, its address much first be "taken", and then loaded in the respective operand, be it a register or memory location.

`sysfilter` leverages the fact that every ELF object in the analysis scope is compiled as `PIC`, in order to identify all AT functions. (`PIC` is mainly used to provide full Address-Space Layout Randomization (ASLR) is enabled by default in modern Linux distributions [90, 91] and incurs negligible performance overhead on x86-64.) Specifically, locations in code, or data, ELF regions that correspond to absolute function addresses must always have accompanying relocation entries (`relocs`), if `PIC` is enabled [91]. This is because these addresses need to be updated with the new, relocated addresses of the respective functions, each time a `PIC`-enabled ELF object is loaded to a different region in the address space.

`sysfilter` begins with identifying all the relocation sections (*i.e.,* sections of type `SHT_-REL` or `SHT_RELA`) in the ELF objects in scope. Next, it processes all the `relocs`, searching for cases where the computation of the relocation involves the starting address of a function (recall that we have already identified the boundaries of every function in scope, during the construction of the DCG). Every such function, whose starting address is used in `relocs` computation, is effectively an AT function. The same function can have its address taken multiple times in different locations (e.g., function arguments, `rvalue` expressions in function bodies, or as part of global `struct`/`union`/C++ object initializers). Relocations that are applied to special sections (e.g., `.plt`, `.dynamic`) are ignored, as they are only related to dynamic binding.

Armed with the set of all AT functions, `sysfilter` proceeds with computing the reachable functions from (each one of) them using the same approach we employed for constructing the DCG, by following targets of direct control flow instructions and resolving PLT entries. The ACG effectively treats the discovered AT functions as "entry points," and then includes every other function that is directly-reachable from them. The combined set of functions in DCG and ACG is a superset of the set of functions in the program's FCG, *i.e.*:

$$V[FCG] \subseteq (V[DCG] \cup V[ACG]) \tag{3.2.1}$$

Quach et al. [16] have previously shown that this approach provides a complete over-approximation to the FCG .

Callgraph construction: Vacuumed call graph Although the combined direct and indirect call graphs, e.g., $DCG \cup ACG$, represents a safe over-approximation of FCG, it is not a tight one: every AT function included in the considered call graph is (potentially) "polluting" it in a considerable manner by bringing in scope every other function that is reachable from itself. In order to keep the over-approximation as tight as possible, `sysfilter` *prunes* the ACG using a technique for software debloating [16, 69]. `sysfilter` terms this optimization as the *vacuumed call graph* (VCG).

Specifically, construction of the VCG considers the location where code pointers for AT functions are created. For code pointers found in code (e.g., `.text`) regions, AT functions can be pruned if their address is only taken form functions that are strictly not part of the already-constructed callgraph. Handling code pointers found in data regions (e.g., `.(ro)data`) is more complex, as code pointers in these sections may be part of nested data structures, the usage of which cannot be tracked statically without debug symbol information. If symbol information is available, the VCG construction process can prune these functions using a similar process. Complete details on the VCG construction process is provided in [111]

In particular, we begin with the observation that each time the address of a function is taken, a code pointer is created. By taking into account the location (ELF section) that such code pointers are created, `sysfilter` further separates those found in code (e.g., `.text`) and data (e.g., `.data`, `.rodata`) regions. For the former, it iterates every function that has been deemed as unreachable, and checks if the address of an AT function is only taken within functions that are (strictly) not part of the call graph. If this condition is true, it removes the respective AT function from ACG, which may result in additional removals (e.g., everything directly-reachable from the removed AT function); `sysfilter` iteratively performs the above until no additional functions can be pruned. While this adds a required component to the analysis beyond a common (stripped) ELF binary, modern toolchains (GCC, LLVM) include symbols in the resulting ELF objects (`.symtab` section) by default, while popular Linux distributions provide symbols for their packaged binaries [112, 113, 114].

The resulting VCG represents a pruned subset of the ACG, specifically:

$$VCG = DCG \cup ACG' \tag{3.2.2}$$

where ACG' denotes the pruned ACG using the approach outlined in this section. Again, VCG is a complete, tight over-approximation of the true FCG: *i.e.*, `sysfilter` only excludes functions that can never be executed by the program (under any possible input); more formally:

$$V[FCG] \subseteq V[VCG] \subseteq (V[DCG] \cup V[ACG]) \tag{3.2.3}$$

Call graph construction example The following example illustrates the construction of each call-graph type from the given program. Figure 3.2 illustrates a C-like program, which we will be using as an example to demonstrate the VCG construction. `sysfilter` will initially include `main` (ln. 19) and `f9` (ln. 5). DCG will also include all the directly-reachable functions from the above initial set: `f1` (reachable from `main`, ln. 22) and `f10` (reachable from `f9`, ln. 22). Thus, at this stage, the callgraph can be represented as:

$$V[DCG] = \{\texttt{main}, \texttt{f1}, \texttt{f9}, \texttt{f10}\} \tag{3.2.4}$$

Next, `sysfilter` will proceed with the construction of the ACG, which, initially, will include all the address-taken functions: `f3` (ln. 14), `f4` (ln. 13), and `f6` and `f7` (ln. 17). ACG will also include all the directly-reachable functions from set of AT functions: `f5` (reachable from `f4`, ln. 10) and `f8` (reachable from `f7`, ln. 7). Thus:

$$V[ACG] = \{\texttt{f3}, \texttt{f4}, \texttt{f5}, \texttt{f6}, \texttt{f7}, \texttt{f8}\} \tag{3.2.5}$$

`sysfilter` will then continue with pruning the ACG as follows. First, it will remove `f4`, as its address is only taken in function `f2` (ln. 13), which is unreachable. This will also result in removing

```
 1 #define ctor __attribute__((constructor))
 2 typedef void (*fptr)(void);
 3
 4 void f10(void) { ... }
 5 ctor void f9(void) { ... f10(); ... }
 6 void f8(void) { ... }
 7 void f7(void) { ... f8(); ... }
 8 void f6(void) { ... }
 9 void f5(void) { ... fp_arr[n](); ... }
10 void f4(void) { ... f5(); ... }
11 void f3(void) { ... }
12
13 fptr f2(void) { ... return &f4; }
14 fptr f1(void) { ... return &f3; }
15
16 fptr fp;
17 fptr fp_arr[] = {&f6, &f7};
18
19 int main(void)
20 {
21   ...
22   fp = f1();
23   ...
24   fp();
25   return EXIT_SUCCESS;
26 }
```

Figure 3.2: VCG construction example. Without symbol information $V[VCG] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f6}, \texttt{f7}, \texttt{f8}, \texttt{f9}, \texttt{f10}\}$, whereas with symbols (or debugging information) available, $V[VCG] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f9}, \texttt{f10}\}$.

`f5`, as it is only directly-reachable from `f4` (ln. 10). If the respective ELF object is stripped [115], the pruning process will terminate at this point, resulting in the following set of functions:

$$V[ACG'] = \{\texttt{f3}, \texttt{f6}, \texttt{f7}, \texttt{f8}\} \tag{3.2.6}$$

If symbol (or debugging) information is available, then `sysfilter` can perform more aggressive pruning by identifying that `fp_arr` is not referenced by any function in scope. (*i.e.*, `fp_arr` is only referenced by `f5`, ln. 9, which has been deemed unreachable). Therefore, the AT functions `f6` and `f7` can also be removed from the ACG, as well as `f8` that is directly-reachable only from `f7` (ln. 7). The net result of the above is the following set of functions:

$$V[ACG''] = \{\texttt{f3}\} \tag{3.2.7}$$

Further details on other nuances associated with callgraph construction, including how `sys-`

`filter` handles GNU `IFUNC` and `NSS` symbols, overlapping code, and hand-written assembly, are described further in [111].

Comparison with existing work As discussed in Chapter 2, other works have used similar methods to statically construct a program's FCG. We believe our analysis provides the tightest and most complete representation using binary analysis. Our callgraph construction process follows Wagner and Dean's [46] over-approximation for handling address-taken functions to compose the ATCG, but their approach requires program source code, and our VCG representation produces a tighter approximation of the callgraph. Giffin et al. [10] use similar methods to Wagner and Dean's approach to analyze SPARC binaries, but their approach does not consider code included in shared libraries.

Confine [63] uses mostly binary static analysis, but relies on source code for analyzing glibc, and its callgraph construction process considers all non-C library functions as reachable. Further, it relies on `objdump` for disassembly, which requires symbol information to precisely identify function boundaries [116]. In contrast, `sysfilter` can produce a tighter FCG, and can operate without symbol information. Zeng et al. [59] uses techniques similar to `sysfilter`, but its handling of indirect function calls relies on points-to analysis, which is inherently incomplete [110], and instead proposes a runtime component to address completeness issues.

3.2.3. System call set construction

The x86-64 ABI dictates that system calls are performed using the `syscall` instruction [117]. `sysfilter` focuses solely on 64-bit applications (*i.e.*, it does not consider syscalls via `int $0x80` or `sysenter`). Moreover, during the invocation of `syscall`, the *system call number* is placed in register `%rax`. Armed with the program's VCG, `sysfilter` constructs the system call set in question as follows.

First, it identifies all reachable functions that include `syscall` instructions, by performing a linear sweep [103] in each function $f \in V[VCG]$ to disassemble reachable code and pinpoint `syscall` instances. Once the set of all the reachable `syscall` instructions is established, `sys-filter` continues with performing a simple *value-tracking* analysis to resolve the exact value(s) of `%rax` on every `syscall` site. The process relies on standard live-variable analysis using *use-define* (UD) chains [118, § 9.2.5]. Specifically, `sysfilter` considers that `syscall` instructions "use" `%rax` and leverages the UD links to find all the instructions that "define" it. In most cases, `%rax` is defined via constant-load instructions (e.g., `mov $0x3, %eax`), and by collecting such instructions and extracting the respective constant values, `sysfilter` can assemble system call sets. If `%rax` is defined via instructions that involve memory operands, `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may be incomplete [109]. The output of the syscall-set extraction component is the collected set of system call numbers in JSON format (see Figure 3.1).

We opt for applying the analysis above in an intra-procedural manner, as our results indicate that this strategy works well in practice. System call invocation is architecture-specific, and typically handled via `libc` using the following pattern (in x86-64):

```
mov $SYS_NR, %eax
syscall
```

where `$SYS_NR = {$0x0, $0x1, ...}`. One exception is the handling of `glibc`'s `syscall()` function [24], which performs system calls indirectly by receiving the respective system call number as argument. If `syscall()` is not-address taken in VCG, then `sysfilter` first identifies the reachable functions that directly-invoke `syscall()`, and performs intra-procedural, value-tracking on register `%rdi` (first argument, system call number). If the address of `syscall()` is taken in the reachable VCG, then `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may be incomplete.

3.2.4. System call set enforcement

The input to the syscall-set enforcement component of `sysfilter` is the set of allowed system calls, as output by the extraction tool, as well as the ELF file that corresponds to the main binary of the application (see Figure 3.1E). Armed with the set of developer-intended syscalls, termed the *syscall policy*, `sysfilter` uses `seccomp-BPF` [77] to enforce it at run-time. The run-time component receives as input a `BPF` "program" [76], passed via `prctl`, or `seccomp`, which is invoked by the kernel on every system call. `BPF` programs are executed in kernel mode by an interpreter for BPF bytecode [76], while just-in-time (JIT) compilation to native code is also supported [50]. Note that the Linux kernel provides support for two different `BPF` variants: (a) classic (`cBPF`) and (b) extended (`eBPF`) [119]. As of Linux kernel v5.8, `seccomp-BPF` makes use of `cBPF` only.

The input to `seccomp-BPF` programs (filters) is a fixed-size `struct` of type `seccomp_data`, passed by the kernel, which contains a snapshot of the system call context: *i.e.,* the syscall number (field `nr`), architecture (field `arch`), as well as the values of the instruction pointer and syscall arguments. `sysfilter` performs filtering based on the value of `nr` as follows: **`if`** $(nr \in \{0, 1, ...\})$

```
 1 #define ARCH AUDIT_ARCH_X86_64
 2 #define NRMAX (X32_SYSCALL_BIT - 1)
 3 #define ALLOW SECCOMP_RET_ALLOW
 4 #define DENY SECCOMP_RET_KILL_PROCESS
 5
 6 struct sock_filter filter[] = {
 7     BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
 8         (offsetof(struct seccomp_data, arch))),
 9     BPF_JUMP(BPF_JMP| BPF_JEQ|BPF_K, ARCH, 0, 7),
10     BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
11         (offsetof(struct seccomp_data, nr))),
12     BPF_JUMP(BPF_JMP|BPF_JGT|BPF_K, NRMAX, 5, 0),
13     BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0, 3, 0),
14     BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1, 2, 0),
15     BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 15, 1, 0),
16     BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 60, 0, 1),
17     BPF_STMT(BPF_RET|BPF_K, ALLOW),
18     BPF_STMT(BPF_RET|BPF_K, DENY)
19 };
```

Figure 3.3: Classic BPF (cBPF) program. Compiled-by `sysfilter`, enforcing the following syscall set: 0 (`read`), 1 (`write`), 15 (`exit`), and 60 (`sigreturn`). The filter checks if the value of field `nr` (syscall number) $\in \{0, 1, 15, 60\}$ via means of linear search.

then ALLOW **else** DENY, where $\{0, 1, ...\}$ is the set of allowed system call numbers. Given such a set, `sysfilter` compiles a `cBPF` filter that implements the above check via means of *linear* or *skip list*-based search. Figure 3.3 depicts a filter that uses the linear search approach to enforce the following set of syscalls: `read` (0), `write` (1), `exit` (15), and `sigreturn` (60). Ln. 7 – 12 implement a standard preamble, which asserts that the architecture is indeed x86-64. This check is crucial as it guarantees that the mapping between the allowed syscall numbers and the syscalls performed is the right one.

To inject the compiled filter, `sysfilter` first generates a dynamic shared object (e.g., `libsysfilter.so`) and links it with the main binary using `patchelf` [120]. The library contains a single function, `install_filter`, registered as a constructor. This directs `ld.so` to automatically load `libsysfilter.so`, and invoke the function during the initialization of the main binary (see Figure 3.1), which installs the filter—and prevents the binary from adding new filters—at load time. For further details on the filter implementation, see [111].

Once the filter is installed, the process process can execute only syscalls specified by the policy. Note that in order to support runtime loading initialization of additional libraries, as well as other loader-related functionality, the system call policy for `ld.so` should also be included in the set of syscalls passed to the enforcement tool. To facilitate this, our enforcement tool accepts a list of policies as arguments, and builds the filter based on the union of all specified policies. (In practice, however, we observe that most binaries already include all system calls used by `ld.so` in their own syscall sets, so this does not represent a significant different in the enforced policies.)

Crucially, our `seccomp-BPF` filters configured as *pinned* to the protected process during its lifetime. Even if the process is completely compromised, attackers cannot remove filters. Since the filtering itself takes place in kernel mode using only the syscall number as input; the syscall arguments are not inspected, and user space memory is not accessed, thereby avoiding the pitfalls related to concurrency and (wrapper-based) syscall filtering [121, 122]. In addition, applications that make use of `seccomp-BPF` are seamlessly supported as well. `BPF` filters are *stackable*, meaning that more than one filter can be attached to a process; if multiple filters exist, the kernel always enforces the most *restrictive* action.

Handling `execve` `sysfilter` prevents enforcement bypasses via the execution of different programs. Specifically, even if a (compromised) process is allowed to invoke `execve`, it still cannot extend its set of allowed syscalls by invoking a different executable that has a (potentially) larger set of allowed syscalls; the same is also true if the process tries to craft a rogue executable in the filesystem, which allows all syscalls (or some of the blocked ones), and execute it. Filter pinning and stacking are essential for ensuring that processes can only reduce their set of allowed syscalls, in accordance to the principle of least privilege [3], but they do interfere with `execve` as they are preserved process attributes.

For example, suppose that programs `P1` and `P2` have the following syscall sets. `P1`: 0 (`read`), 1 (`write`), 15 (`exit`), and 59 (`execve`); `P2`: 0 (`read`), 1 (`write`), 2 (`open`), 3 (`close`), 8 (`lseek`) ,9 (`mmap`), 11 (`munmap`), and 15 (`exit`). If `P2` is invoked normally, then it will operate successfully. However, if `P2` is invoked via `P1`, then the resulting process will not be able to issue any other syscall than `read`, `write`, `exit`, and `execve` (the last two are not even required by `P2`).

To deal with this issue `sysfilter` can be configured to enforce `P1` with the *union* of the syscall sets of `P1`, and `P2`. In this case, both programs will function correctly. However, this over-privileges `P1`. as it must have access to the additional syscalls required by `P2` (in this case, `lseek`, `mmap`, and `munmap`).

Given these limitations, `sysfilter` is not geared towards sandboxing applications that invoke *arbitrary* scripts or programs (e.g., command-line interpreters, managed runtime environments); other schemes, like Hails [123], SHILL [124], and the Web API Manager [125], are better suited for this task. Exploring ways to handle `execve` while safely permitting changes in filters is ongoing work. As one example, our preliminary results (§ 4.3) indicate that combined policy creation can be automated to some extent by employing static value-tracking analysis to resolve the first argument of `execve` calls.

3.3. Prototype implementation

`sysfilter` is built atop the Egalito framework [90]. Egalito is a binary *recompiler*; it allows rewriting binaries in-place by first lifting binary code into a layout-agnostic, machine-specific intermediate representation (IR), dubbed EIR, and then allowing "tools" to inspect or alter it, as a way to simplify tasks such as for instrumentation, hardening, or patching.

We implemented `sysfilter` as an Egalito "pass" (in C/C++), which creates the analysis scope and constructs the VCG (or can stop at construction of the DCG or ACG, if desired). Note that we do not utilize the binary rewriting features of Egalito; we only leverage the framework's API to precisely disassemble the corresponding binaries and lift their code in EIR form, which, in turn, we use for implementing the analyses required for constructing the DCG, identifying all AT functions for building the ACG, pruning unreachable parts of the call graph for assembling the VCG. We chose Egalito over similar frameworks as it employs the best jump table analysis to date. During the development of our prototype tools we also improved Egalito by adding better support for hand-coded assembly, fixing various symbol resolution issues, and re-architecting parts of the framework to reduce memory pressure, and upstreamed all our changes.

Our prototype implementation consists of ≈2.5 KLOC of C/C++ and ≈150 LOC of Python, along with various shell scripts (glue code; ≈120 LOC). The C/C++ component uses the Egalito framework [90]. The extraction tool is a further set of Egalito "passes" on top of `sysfilter`, which identifies identifying `syscall` instructions in all VCG functions, and performs value tracking.

The enforcement tool is implemented in Python and is responsible for generating the `cBPF` filter(s), and `libsysfilter.so`, and attaching the latter to the main binary using `patchelf`. Both tools are open-source and available at `https://gitlab.com/brown-ssl/sysfilter`.

3.4. Evaluation: correctness and performance

We begin our evaluation of `sysfilter` by investigating its correctness and runtime performance. In § 4.3, we continue this evaluation with a detailed study of its effectiveness based on our initial large-scale experiments when running `sysfilter` on ≈30K binaries in Debian `sid`.

Application		Version	Syscalls	Tests	Pass?
FFmpeg	(124)	4.2	167	3756	✓
GNU Core.	(100)	8.31	148	672	✓
GNU M4	(1)	1.14	70	236	✓
MariaDB	(156)	10.3	153	2059	✓
Nginx	(1)	1.16	128	356	✓
Redis	(6)	5.0	104	81	✓
SPEC CINT.	(12)	1.2	82	12	✓
SQLite	(7)	3.31	139	31190	✓
Vim	(3)	8.2	163	255	✓
GNU Wget	(1)	1.20	107	130	✓

Table 3.1: Correctness test. The numbers in parentheses count the different binaries included in the application/package. "Syscalls" indicates the number of system calls in the allowed set; in case of applications with multiple binaries that number corresponds to the unique syscalls across the syscall sets of all binaries in the package. "Tests" denotes the number of tests run from the validation suite of the application.

Testbed Our experiments in this section were performed on an 8-core Intel Xeon W-2145 3.7GHz CPU, armed with 64GB of (DDR4) RAM, running Devuan Linux (v2.1, kernel v4.16). All applications (except SPEC CINT2006) were obtained from Debian `sid`, which is the development (unstable) distribution of Debian Linux [126], as it provides the latest versions of upstream packages along with debug/symbol information [113].

3.4.1. Correctness

We used 411 binaries from various packages/projects with `sysfilter`, including GNU Coreutils, Nginx, Redis, SPEC CINT2006, SQLite, FFmpeg, MariaDB, Vim, GNU M4, and GNU Wget, to extract and enforce their corresponding syscall sets. The results are shown in Table 3.1. Once sandboxed, we stress-tested them with ≈37.3K tests from the projects' validation suites. We did not include tests that required the application to execute arbitrary external programs, such as tests with arbitrary commands used in Vim scripts, Perl scripts in Nginx, and arbitrary shell scripts to load data in SQLite and M4. In all cases, `sysfilter` managed to extract a complete and tight over-approximation of the respective syscall sets, demonstrating that our prototype can successfully handle complex, real-world software.

3.4.2. Performance

To assess the run-time performance impact of `sysfilter`, we used SPEC CINT2006 benchmarking suite, Nginx web server, and Redis data store—*i.e.,* 19 program binaries in total; the selected binaries represent the most performance-sensitive applications in our set and are well-

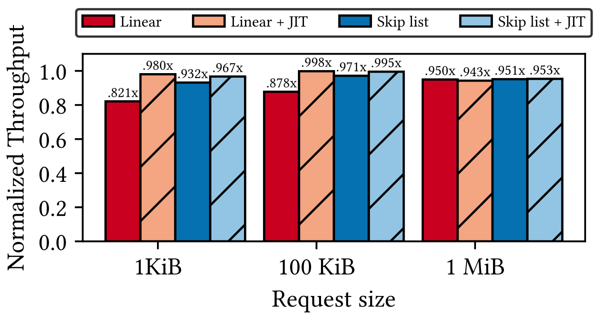


Figure 3.4: Impact of `sysfilter` on Nginx.

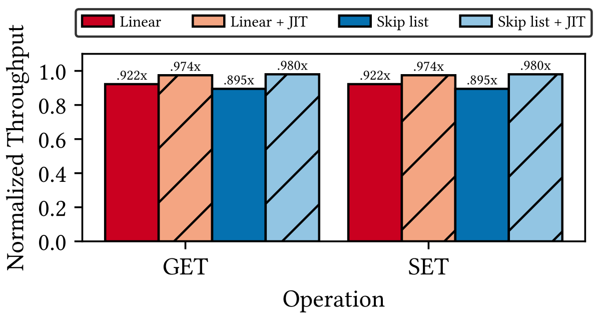


Figure 3.5: Impact of `sysfilter` on Redis.

suited for demonstrating the relative overhead(s). We also explored different settings and configurations, including interpreted vs. JIT-compiled `BPF` filters, and filter code that implements sandboxing using a linear search vs. filter code that utilizes a skip list-based approach (Sec. 3.2.4). In the case of SPEC, we observed a run-time slowdown of ≤1% under all conditions and search methods.

In our experiments, we disable the default `seccomp` filter flag `SECCOMP_FILTER_FLAG_-SPEC_ALLOW` [49] to disable the speculative store bypass (SSB) mitigation. This is a configurable option in our enforcement tool—however, the SSB mitigation is only relevant when `BPF` programs of *unknown provenance* are loaded in kernel space to further assist mounting Spectre attacks [127] (variant 4 [128])—`sysfilter` `cBPF` programs are not malicious nor attacker-controlled. When the mititation is enabled, we observe that it incurs an additional ≈10% overhead in all cases.

Figure 3.4 and Figure 3.5 illustrate the impact of `sysfilter` on Nginx (128) and Redis (103)—the numbers in parentheses indicate the corresponding syscall set sizes, while "Binary" corresponds to skip list-based filters. We configured Nginx to use 4 worker processes and measured its throughput using the `wrk` tool [129], generating requests via the loopback interface from 4 threads, over 256 simultaneous connections, for 1 minute. Overall, `sysfilter` diminishes reduction in throughput by using skip list-based filters (compared to linear search-based ones) when JIT is disabled, with maximum reductions in throughput of 18% and 7%, respectively. The differences in compilation strategy appear to be normalized by jitting, which showed a maximum drop in throughput of 6% in all conditions. We evaluated Redis similarly, using the `memtier` tool [130], performing a mix of `SET` and `GET` requests with a 1:10 request ratio for 32 byte data elements. The requests were issued from 4 worker threads with 128 simultaneous connections, per thread, for 1 minute. `sysfilter` incurs maximum throughput reductions of 11% and 3%, with and without JIT support, respectively. Considering everything, the sandboxed versions of SPEC, Nginx, and

Redis, exhibited minimal or negligible run-time slowdown—suggesting that `sysfilter` can be readily used in real-world settings and deployments.

Chapter 4

Reducing syscall overprivilege at scale

4.1. Measuring overprilege at scale

A primary goal of our work is to identify effective to reduce overprivilege for a wide variety of programs. In order to address this challenge, we conduct a series of large-scale studies to characterize overprivilege—and the extent to which our techniques can help reduce it—across a whole Linux distribution. Broadly, our studies have three goals:

1. To test our techniques on a large set of binaries, highlighting edge cases where static disassembly may fail
2. To characterize system call overprivilege at scale, motivating the need for tools to reduce overprivilege in a wide variety of programs
3. To demonstrate the effectiveness of our techniques by showing how their application across a diverse set of programs can improve the security posture of a system

To conduct our studies, we built `autopkg`, a framework to run our tools on all ELF application binaries in a Linux system. We chose to use Debian Linux [126] for our studies. Debian is a large, well-established distribution—at the time of this writing, it contains over 50K software packages for the x86-64 architecture. Specifically, we chose to use Debian `sid`, the development branch, due to its high availability od debug symbol packages [113]. To analyze each package, we implemented `autopkg`, a custom framework to download and install each package in the repository, locate ELF binaries in each package, and run our tools on each binary to gather information about syscall API usage and library code bloat. The following sections describe the `autopkg` framework and present preliminary results gathered from analysis of over ≈30K binaries.

Related studies We are aware of two similar distribution-wide studies that have been conducted in the past. Tsai et al. [131] performed a similar study to ours (on binaries in Ubuntu v15.04) to characterize the usage of the syscall API, as well as that of `ioctl`, `fcntl`, `prctl`, and pseudo-filesystem APIs. Their study focuses on quantifying API complexity and security-related usage trends, such as unused syscalls and adoption of secure APIs over the legacy ones. Our study of the syscall API is focused on evaluation of our syscall extraction tool. In antithesis to our FCG construction tools, the call graph construction approach of Tsai et al. does not consider initialization/-finalization code nor does it identify AT functions that are part of global `struct`/`union`/C++ object initializers. Prior to their work on piecewise compilation, Quach et al. [132] perform a study of code bloat across 12 large applications (web browsers, text editors, media players, etc.) on Debian Wheezy, using static analysis to construct an FCG similar to our ACG (§ 3.2.2). Our study explores a much wider range of binaries, and presents a more precise analysis of function usage with our more concise VCG. Overall, we consider this work complementary, and focus on making the analysis more scalable, precise, and complete.

4.2. `autopkg` design

Broadly, the goal of our `autopkg` framework is to run an analysis tool (*i.e.*, `sysfilter` or `libfilter`) on every x86-64 ELF binary in the repository. At the time of writing, Debian `sid` contains over 50K packages for the x86-64 architecture, distributed over three major repositories: `main`, `contrib`, and `nonfree`. In Debian, a *package* represents a single installable software unit: one package may contain multiple ELF *binaries* that require analysis (or may contain none, in the case of, e.g., packages containing only documentation). Each package has a list of *dependencies*, which specify the shared libraries required by its binaries. Analyzing each binary in the distribution is a challenging task. In order to correctly resolve shared library loading, packages, their dependent libraries, and (to support our analysis) all required debug symbols, must be installed to their canonical paths as on a normal system. Further, not all packages can be installed at the same time due to dependency conflicts, necessitating a more segmented approach. To meet these challenges, we developed `autopkg` to automate the installation and analysis of a large set of binaries in Debian.

Figure 4.1 depicts the architecture of `autopkg`. The analysis process has two steps: an offline phase to generate a list of candidate packages, and a distributed, online phase where a coordinator dispatches package analysis jobs to a set of workers that install and process individual packages.

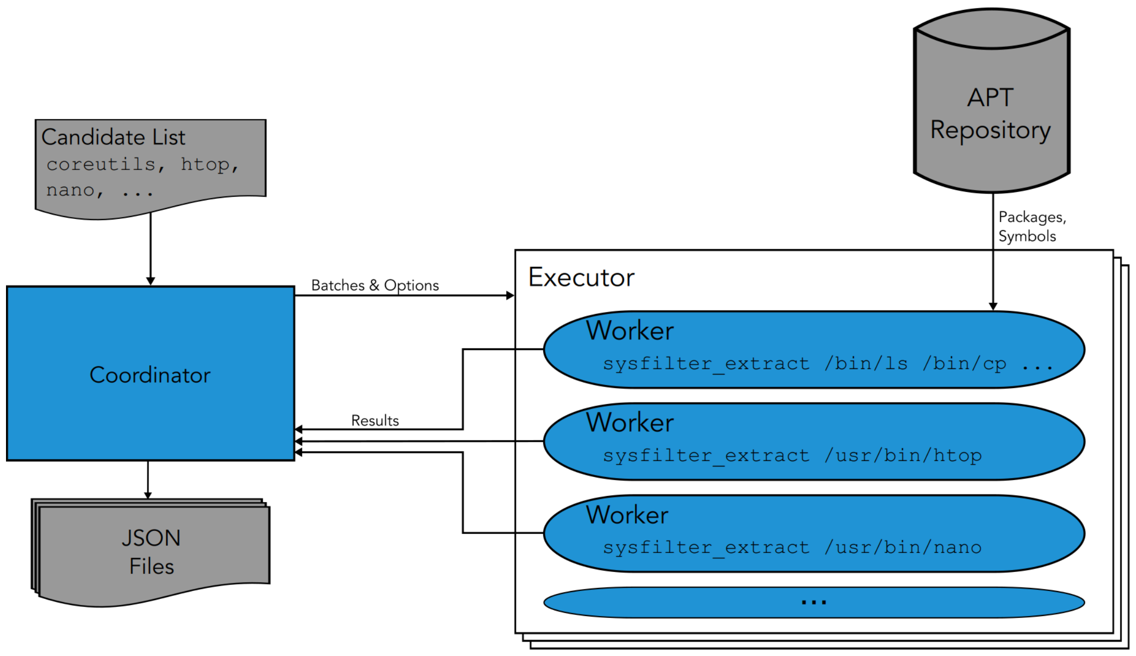

APT Repository
Candidate List
coreutils, htop, nano, ...
Packages, Symbols
Batches & Options
Executor
Coordinator
Worker
sysfilter_extract /bin/ls /bin/cp ...
Results
Worker
sysfilter_extract /usr/bin/htop
Worker
sysfilter_extract /usr/bin/nano
JSON Files
...

Figure 4.1: `autopkg` architecture.

4.2.1. Identifying package candidates

The first step in our analysis involves identifying the list of *candidate packages* that contain executable C/C++ binaries. We consider packages from the three major Debian repositories, namely `main`, `contrib`, and `nonfree`. To build a list of candidate packages, we excluded packages that did not contain executable code by definition, including: documentation packages (`*-doc`); development headers (`*-dev`); debug symbols (`*-dbg`, `*-dbgsym`); metapackages and virtual packages; architecture-agnostic packages (architecture type '`all`'), which cannot include x86-64 binaries; kernel packages; and packages that contain only shared libraries. Note that shared libraries and other excluded packages can be installed during processing as dependencies of candidate packages.

Before beginning analysis, we divide the candidate packages into *batches* of at least one package. A batch represents a group of packages processed by a worker at one time, and therefore must represent a set of packages that can be installed simultaneously. To construct a batch, we attempt a "dry-run" installation of a package to identify other packages that will be installed at the same time. If any of the installed packages are in the list of candidates, they are included in that batch for processing as well.[1]

[1]While we tested using larger batch sizes, we found this did not improve overall performance due to the high potential for installation errors and conflicts.

The resulting *batch list* provides the list of jobs to dispatch to worker processes. Since Debian `sid` is an unstable distribution, the list of available packages changes continuously, so we perform this step at the start of each analysis run. As analyses are completed, we also update the list of excluded packages with packages that were found to contain no x86-64 binaries during processing: these usually include packages containing scripts, modules for scripting languages (e.g., Python, Perl, Ruby), or non-code assets like icon sets.

4.2.2. Package installation and analysis

Once the batch list is prepared, `autopkg` dispatches batches among a set of worker processes. Critically, our framework uses containers to isolate and automate installations for each package. A template container image is prepared using `debootstrap` [133] containing a base install of Debian sid and a curated list of common packages and symbol packages. Each worker is spawned inside a new container built from the template image using an overlay filesystem, thus isolating any filesystem changes from other worker invocations. The worker process installs the package and its dependencies, scans the installed files for suitable binaries, identifies and installs required symbol packages, and, finally, runs the analysis tool using the configuration specified by the coordinator. When the process is complete, the worker sends the results back to the coordinator and then exits. Upon exiting, the container and its filesystem overlay are deleted. While this means that workers may spend significant time installing packages for large applications, we opted for this "clean slate" approach to reduce error handling associated with dependency conflicts.

`autopkg` is built using a distributed architecture to allow for distribution of batches across multiple analysis machines. At the start of each analysis run, a central *coordinator* process is invoked with the batch list and a configuration file specifying the analysis tool and any configuration arguments required to run it. Each analysis machine runs an *executor* process, responsible for launching a configurable number of workers (usually one per CPU core). When a worker slot is free, the executor requests a batch from the coordinator and launches a new container with the worker process. To ensure no memory contention between workers, workers are spawned with fixed memory limits out of a total allocation for the system.

Our implementation of `autopkg` is built with ≈4.4KLOC of Python and uses `python-apt` [134] as an interface to Debian's `libapt` to install packages and track dependencies. We use Podman [56] to run container images and enforce memory limits. All interprocess communication occurs using gRPC [135]. While `autopkg`'s architecture supports distribution of batches on multiple machines, for our current studies we have found it sufficient to run on a single machine.

We note that a significant portion of `autopkg`'s code relates to error handling. Since Debian `sid` is an unstable distribution containing developers' latest package changes, the subset of packages in the repository that can be processed at a given moment varies continuously. Broken packages are common, and some packages may not be installable due to missing or outdated dependencies. We typically find that 2–5% of the packages fail to install in a given analysis run. Amusingly, during one experiment, an update to the package containing `libc.so` caused installations to fail for *all* packages—the problem was resolved by the Debian maintainers within a few hours. For less catastrophic errors, identifying ways to improve resolution of certain install errors is an area of ongoing work.

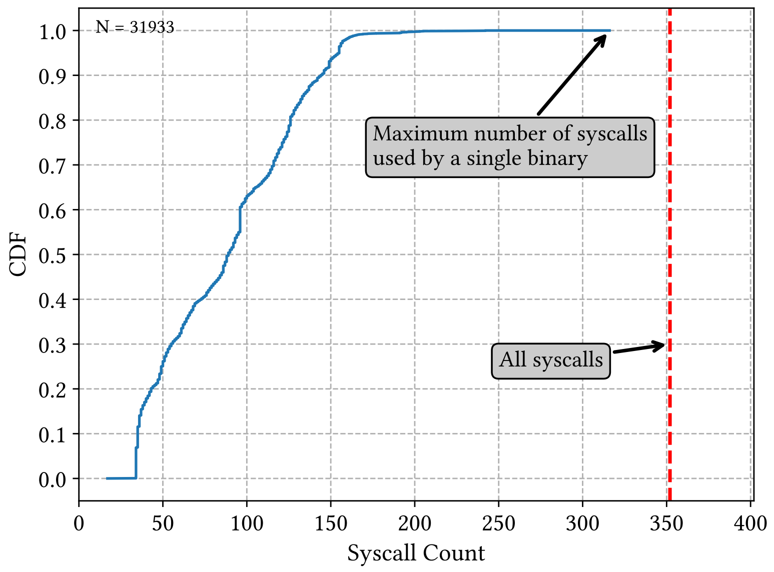
We performed our analysis runs from a single host, armed with an 8-core AMD Ryzen 2700X 3.7GHz CPU with 64GB of RAM, running Arch Linux (kernel v5.2). This section presents results from analysis runs for `sysfilter` conducted in July 2021. We have performed multiple analysis runs with each tool and achieved broadly similar results across runs, demonstrating (relative) stability in our analysis procedure given the variability in package stability in `sid`. We also conduct similar analyses of code bloat using our `libfilter` tool, which we discuss in § 6.2.

4.3. System call usage in the wild

In this study, we processed a total of 34781 binaries across 9024 packages, 31659 (91%) of which could be analyzed successfully. For this study, we configured `sysfilter` to report syscall usage for each FCG type described in § 3.2.2: the direct call graph (DCG), address-taken call graph (ACG) and the vacuumed call graph (VCG). Except where otherwise noted, all reported data is based on the VCG representation.

4.3.1. Syscall set size per binary

Across all binaries, we observe that nearly all of the system call API is utilized, but many binaries only use a small portion of the syscall API. Out of the total set of 352 syscalls (as of kernel v5.8) we observe 334 syscalls utilized across all ≈31.6K binaries, which represents 94.8% of the total set. Figure 4.1 shows the distribution of the number of syscalls used by each binary processed. Overall, we observe that this distribution has a long tail, with a few binaries using a large number of syscalls—Table 4.1 shows the top 10 binaries by syscall set size. The application using the most syscalls is `stress-ng` a stress-testing tool designed to exercise hardware and OS functionality, which uses 316 syscalls, or 90% of the total syscall API. The median syscall count per



Percentile	Syscalls (% of API)
Min	17 (5%)
50th	89 (25%)
90th	144 (41%)
95th	155 (44%)
99th	166 (47%)
99.5th	192 (55%)
Max	316 (90%)

Figure 4.2: Distribution of syscall set size across binaries on Debian `sid`.

binary is 89 syscalls, with the 95th percentile at only 155 syscalls—thus, 50% of binaries use $\leq$50% of the syscall API, with 95% of binaries using at most 47% of the API. This demonstrates that most binaries in the distribution are *significantly* overprivileged in terms of syscall API access, and can thus benefit from reducing their privileges.

With one exception, we find that *all* binaries processed use at least 40 syscalls. The sole exception is the statically-linked binary `mtcp_restart` that in package `dmctp` (which provides distributed, multi-threaded checkpointing) uses only 17 syscalls—this binary performs syscalls directly, without using any library wrappers.

In the general case, even the simplest of programs, such as a program equivalent to `/bin/-false`, utilize 40 syscalls due to the paths included by initialization functions that load shared libraries: e.g., `mmap` and `mprotect` are ubiquitous as they are always reachable from `_start`, even before `main` is invoked.

Binaries that make arbitrary syscalls We found 105 binaries (0.55% of those analyzed) where `sysfilter` was unable to construct the complete syscall sets. Many of these binaries are found in the programs using the highest number of syscalls, as listed in Table 4.1. These programs contain code where the syscall number is not defined as a constant but instead determined at runtime, so `sysfilter`'s static value tracking approach (§ 3.2.3) cannot determine a constant value. On manual examination, we found that these cases are fairly isolated. Specifically, the Qemu vir-

Package	**Binary**	**Syscalls (% of API)**
stress-ng	`stress-ng`	316 (90%)†
qemu-user	`qemu-*` (30 binaries)	242 (69%)†
linux-perf-5.10	`perf_5`	219 (62%)†
vim-gtk3	`vim`	214 (61%)†
vim-athena	`vim`	210 (60%)†
vim-nox	`vim`	208 (59%)†
aoflagger	`rfigui`	208 (59%)
radare2-cutter	`Cutter`	207 (59%)
gnubg	`gnubg`	206 (58%)
profanity	`profanity`	205 (58%)

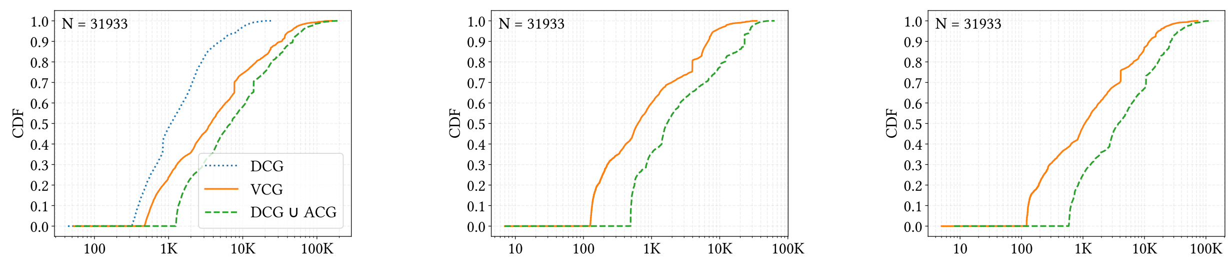
Table 4.1: Binaries using the most syscalls. Binaries marked with a (†) can also make arbitrary system calls.

tualization framework and `stress-ng` stress-testing tool contain application-specific, arbitrary syscall dispatchers (like `glibc`'s `syscall()`), which is expected given their functionality. We also find binaries that link with scripting language runtimes like `libperl` and `libruby` (such as `vim`), which also contain arbitrary system call dispatchers. Otherwise, we find that syscall sites follow the pattern '`mov $constant, %eax; syscall`', supporting our static value tracking approach.

Dynamically-loaded code (`dlopen`/`dlsym`) and `execve` We further employ our value-tracking approach to investigate the impact of `execve` as well as code loaded dynamically with `dlopen` and `dlsym`. `sysfilter` can resolve ≈89% of all `dlsym` arguments, ≈45% of all `dlopen` arguments, and ≈19% of all `execve` arguments. We observed several cases in common libraries where value-tracking fails, which may benefit from special-case handling handling: e.g., ≈20% of `dlopen/dlsym` usages relate to NSS functionality, while ≈22% relate to OpenGL APIs. Code loaded dynamically with `dlopen/dlsym` represents a case where `sysfilter` (or *any* static analysis tool) may provide an incomplete analysis, since the libraries and functions loaded may be dependent on program input (such as configuration files). We discuss methods for handling common-case usages of `dlopen/dlsym` in Chapter 5 and provide an analysis of the maximum impact on syscall sets.

4.3.2. System call invocation sites

For each system call site, we record the ELF file and function name that contained the syscall, which provides insights into how programs invoke system calls. Nearly all programs invoke most system calls via the standard C library, `glibc`. We find that 162 syscalls, 48.5% of those observed,



(a) Total functions **(b)** Funcs. with indirect callsites **(c)** Address-taken functions

Figure 4.3: Distribution of functions in extracted callgraphs. Note that the DCG only contains direct function calls, and thus does not track indirect calls or address-taken functions.

are invoked *only* from `glibc`. Conversely, 77 syscalls are *never* made via `glibc`, and are instead invoked directly by binaries or other shared libraries, most commonly `set_tid_address` and `set_robust_list`, which do not have `glibc` wrappers.

4.3.3. Effectiveness of FCG approximation

Figure 4.3 shows the number of functions using the three FCG approximation methods (§ 3.2.2): DCG, $DCG \cup ACG$, and VCG. For all binaries, the total number of functions in the VCG (shown in Figure 4.3a) is always in between DCG and $DCG \cup ACG$, with ≈85% fewer functions observed at the median and ≈42% fewer at the 90th percentile. Similarly, we observe that the VCG reduces the number of functions with indirect callsites and AT functions in the callgraph by an order of magnitude, demonstrating the effectiveness of the VCG's additional pruning.

We also examine the number of syscalls reported for each FCG type. Figure 4.4 shows the number of syscalls we extract from each binary sorted by the count for the VCG. Each binary represents three points on the figure (*i.e.,* one for each method). For all binaries, the count for the VCG is always in between that of DCG and $DCG \cup ACG$. Thus, for our dataset, VCG represents a safe, *tight* over-approximation of the FCG.

4.3.4. Effectiveness of syscall restrictions

To assess the security impact of syscall filtering, we investigated how `sysfilter` reduces the attack surface of the OS kernel, by inquiring what percentage of all C/C++ applications in Debian `sid` (≈30K main binaries) can exploit 23 (publicly-known) Linux kernel vulnerabilities—ranging from memory disclosure and corruption to direct privilege escalation—even after hardened with `sysfilter`. A list of the number of binaries in our dataset affected by each CVE is shown in Table 4.2. Depending on the exact vulnerability, the percentage of binaries that can still attack the

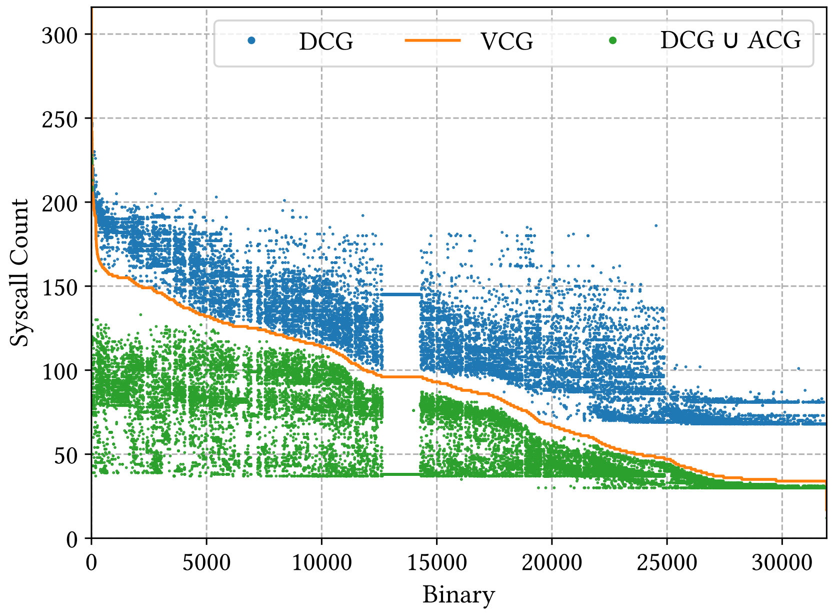


Figure 4.4: Syscall count for different FCG construction methods. The number of syscalls reported for each binary is shown, sorted by the count for the VCG.

kernel ranges from 0.2% – 67.19%. Although `sysfilter` does not defend against particular types of attacks (e.g., control- or data-flow hijacking [78]), our results demonstrate that it can mitigate real-life threats by means of least privilege enforcement and OS attack surface reduction. For example, we observed 152 binaries (only 0.48% of those analyzed) using the `ptrace` syscall—the remainder of the binaries in the study have the ability to invoke `ptrace`, even though they do not use it. By restricting other binaries from invoking `ptrace`, we reduce the kernel's attack surface by limiting access to this interface.

4.3.5. Analysis performance and scalability

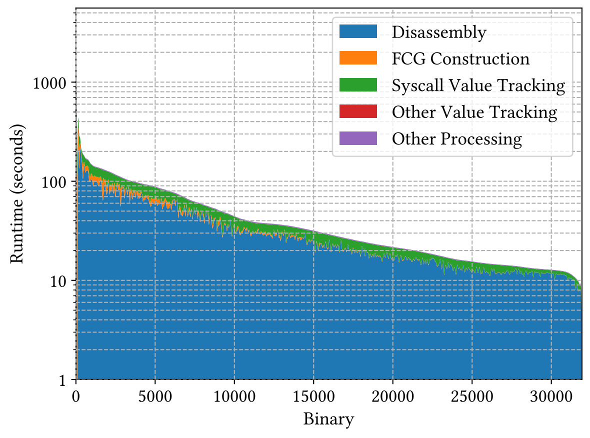
To collect data for our studies, we configured `sysfilter` to collect system call sets for four separate FCG passes, the three mentioned in this section and the ATCG, which we discuss further in Section 5.2. In this configuration, the median runtime is about 70s per binary, with 90% of binaries completing in 300s. For a single pass that generates only the VCG, which would be a more suitable configuration in production, the median runtime is reduced to about 30s per binary, with 90% of binaries completing within 105 seconds. In either case, however, we believe that this demonstrates `sysfilter`'s scalability: `sysfilter` can *automatically* analyze each binary in the Debian `sid` distribution (e.g., ≈30K binaries) in a relatively short time (*i.e.*, 1-2 minutes for the

CVE	Syscalls Involved	Vulnerability Type	Binaries (%)
CVE-2019-11815	`clone, unshare`	Memory corruption	21455 (67.19)
CVE-2013-1959	`write`	Direct privilege escalation	21455 (67.19)
CVE-2015-8543	`socket`	Number overflow leads to type confusion	20190 (63.23)
CVE-2017-17712	`sendto`, `sendmgs`	Memory corruption	20102 (62.95)
CVE-2013-1979	`recvfrom`, `recvmsg`	Direct privilege escalation	19848 (62.16)
CVE-2017-18509	`setsockopt`, `getsockopt`	Memory corruption	19810 (62.04)
CVE-2021-22555	`setsockopt`	Memory corruption	19801 (62.01)
CVE-2016-4997	`setsockopt`	Memory corruption	19801 (62.01)
CVE-2016-4998	`setsockopt`	Memory disclosure	19801 (62.01)
CVE-2018-14634	`execve`, `execveat`	Memory corruption on suid program	13870 (43.43)
CVE-2017-14954	`waitid`	Memory disclosure	13171 (41.25)
CVE-2014-9529	`keyctl`	Memory corruption	4468 (13.99)
CVE-2016-0728	`keyctl`	Memory corruption	4468 (13.99)
CVE-2018-12233	`setxattr`	Memory corruption	2605 (8.16)
CVE-2014-5207	`mount`	Direct privilege escalation	367 (1.15)
CVE-2019-13272	`ptrace`	Direct privilege escalation	154 (0.48)
CVE-2018-1000199	`ptrace`	Memory corruption	154 (0.48)
CVE-2014-4699	`fork`, `clone`, `ptrace`	Register corruption	119 (0.37)
CVE-2017-6001	`perf_event_open`	Direct privilege escalation	81 (0.25)
CVE-2014-7970	`pivot_root`	DoS	79 (0.25)
CVE-2019-10125	`io_submit`	Memory corruption	73 (0.23)
CVE-2016-2383	`bpf`	Memory corruption	65 (0.20)

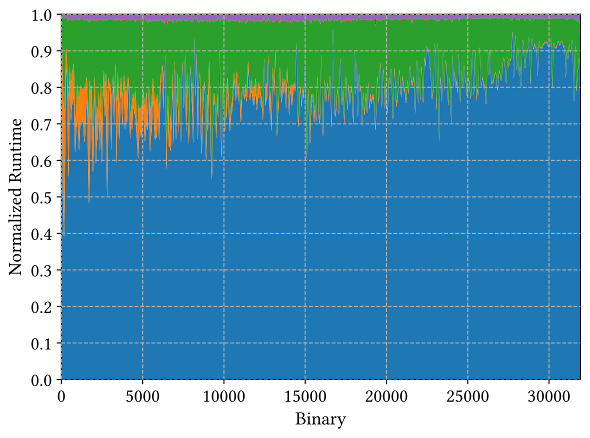
Table 4.2: Effectiveness analysis. The column "Binaries" indicates the number (and percentage) of binaries observed in the large scale analysis on Debian `sid` applications that use the system calls related to the respective vulnerability. (Underlined entries correspond to vulnerabilities that involve namespaces.)

VCG configuration). This represents a significant reduction in cost compared to approaches that require runtime tracing or source code analysis, which would require specific configuration and testing for each individual application.

While analyzing each binary, we measured the time for each stage of processing: disassembly to construct the analysis scope, FCG construction, syscall number extraction, and additional value tracking studies, such as collecting `dlsym` and `execve` usage (§ 4.3.1). Figure 4.5 shows the runtime of our extraction tool to disassemble all binaries and construct just the VCG for each analysis phase (excluding the times for the other callgraph passes used only for testing). The most dominant phase is the time to disassemble the binary and all its shared libraries, compromising ≈70% or more of the analysis time for most binaries. The next highest phase was system call set extraction (§ 3.2.3), comprising ≈10–20% of the analysis time, due to the required data-flow tracking. We believe these areas are opportunities for improving the tool's performance, as well as that of the Egalito framework, to further improve our analysis.



(a) Runtime per binary



(b) Normalized

Figure 4.5: Extraction tool runtime. Runtime is shown for the different phases of analysis to construct the VCG for each binary, sorted by the largest runtime. Points are smoothed to show the overall distribution between analysis phases. The phase "Other processing" refers to non-analysis tasks such as startup, output generation, and cleanup.

4.4. Inferring policies for container images

We used `sysfilter` to identify system call policies for sets of binaries in container images. Container runtimes like Docker [55] and Podman [56] automatically apply a configurable `seccomp-BPF` policy when running containers to restrict the system calls they can invoke, which is critical for preventing containers from bypassing the containerized environment. The default Docker policy permits 287 syscalls for standard containers and 306 syscalls for privileged containers, *i.e.*, 74% and 78% of the total syscall API, respectively, making this policy very coarse. Indeed, we find only one binary in our study (`stress-ng`) that requires more than 287 syscalls.

Accordingly, we examine to what extent `sysfilter` can be used to infer more-restrictive system call policies by reporting on the syscall sets we identified in common sets of Linux packages in popular container images published in the Docker Hub image repository [136]. For this example analysis, we do not analyze Docker container images directly—rather, we leverage our existing dataset and `autopkg` infrastructure for processing Debain `sid` packages to provide an equivalent analysis. Using `sysfilter` to analyze binaries inside container images is feasible with additional engineering effort.

We selected 10 official images for popular C/C++ applications, each with >10M downloads from the Docker Hub. To simplify the process of translating packages in the container's base distribution to Debian `sid` package names, we select container images that were build from dis-

Image	#Pkg	#Bin	Default Policy (287 total)			Privileged (306 total)		
			Added	Removed	Reduced Policy	Added	Removed	Reduced Policy
debian	91	321	21 (7%)	72 (25%)	215 (-25%)	11 (4%)	81 (28%)	225 (-21%)
haproxy	89	305	17 (6%)	72 (25%)	215 (-25%)	10 (3%)	84 (29%)	222 (-22%)
httpd	113	314	17 (6%)	72 (25%)	215 (-25%)	10 (3%)	84 (29%)	222 (-22%)
mariadb	178	412	21 (7%)	63 (22%)	224 (-21%)	11 (4%)	72 (25%)	234 (-18%)
memcached	90	304	17 (6%)	70 (24%)	217 (-24%)	10 (3%)	82 (29%)	224 (-21%)
nginx	134	309	17 (6%)	69 (24%)	218 (-24%)	10 (3%)	81 (28%)	225 (-21%)
postgres	137	360	17 (6%)	72 (25%)	215 (-25%)	10 (3%)	84 (29%)	222 (-22%)
redis	88	308	17 (6%)	70 (24%)	217 (-24%)	10 (3%)	82 (29%)	224 (-21%)
spiped	89	307	17 (6%)	73 (25%)	214 (-25%)	10 (3%)	85 (30%)	221 (-22%)
ubuntu	92	324	17 (6%)	70 (24%)	217 (-24%)	10 (3%)	82 (29%)	224 (-21%)
Mean			18 (6%)	70 (24%)	217 (-24%)	10 (3%)	82 (27%)	224 (-26%)

Table 4.3: System call sets for container images. "Added" denotes the number of syscalls identified by `sysfilter` not present in Docker's policy. "Removed" denotes the number of syscalls permitted by the policy but not found in the container image. "Reduced policy" denotes the size of the container policy without the "removed" syscalls, *i.e.*, a policy minimized to the specific container using `sysfilter`. Percentages are relative to the size of the policy.

tributions using the `apt` package manager. For each container image, we mapped each binary to its providing package using `dpkg` and then looked up the equivalent `sid` package in our dataset. Across all images, we considered a total of 260 unique packages, all but 20 of which could be resolved automatically. We manually resolved the remainder, which differed only be version numbers (e.g., `mariadb-server-10.5` vs. `mariadb-server-10.4`). While this mapping between Linux distributions does not provide an exact representation of each container's syscall sets, it does provide a *tight* approximation of the syscall sets observed across a commonly-installed set of packages in popular Docker images.

Table 4.3 shows the total syscall set size that `sysfilter` identified for each image, compared to the default container policy. The `mariadb` image had the largest syscall set, with 245 syscalls identified. We compare the syscall set sizes reported for each image against two container policies: the default policy, and the policy for containers in "privileged" mode—the latter grants `CAP_SYS_ADMIN` to the container and includes 10 additional syscalls. We observe that `sysfilter` can provide further privilege reductions beyond the fixed container policy by identifying system calls that are unused across all binaries in the image, with an average of 70 syscalls removed from the default policy, and 82 syscalls removed from the privileged policy, reducing the policy size by 24% and 27%, respectively.

Interestingly, `sysfilter` also identifies a small number of syscalls that applications in a container may invoke that are *not* permitted by either container policy. We find that these system calls result from features of common system utilities that are not applicable inside a container. For example, the GNU coreutils utilities `hostname` and `date`, frequently used to fetch the system hostname and time, respectively, also have features to *set* the hostname and time (using `sethost-name` and `clock_settime`, respectively). Since altering these system features is not permitted inside a container by default, the policy blocks these syscalls—and a benign container should never invoke them. This is evidence of *code bloat* inside the container, indicating an opportunity for software debloating tools to remove unnecessary code (cf. § 2.2.2, 6.2). Nevertheless, even on bloated images, we believe that `sysfilter` demonstrates utility for reducing container privileges atop existing fixed container security policies.

Chapter 5

Improvements to callgraph precision

Our work with `sysfilter` demonstrated the feasibility and effectiveness of using static analysis techniques to reduce overprivilege in terms of system call sets. Our distribution-wide studies on Debian `sid` have illustrated the need for reducing system call overprivilege and demonstrated our ability to analyze a wide range of programs. In addition, our large-scale studies, comprising $\approx$30K binaries, provide a significant corpus for studying potential enhancements to our analysis techniques that can benefit a large number of programs. Accordingly, we leverage our analysis tools and our dataset to investigate two key challenges:

1. Characterizing the impact code loaded dynamically using `dlopen`/`dlsym`
2. Further reducing privileges *within an application* by identifying useful partitions in the callgraph

The following sections discuss our efforts to explore each challenge and evaluate the extent to which we can improve the scalability and effectiveness of our techniques.

5.1. Handling dynamically-loaded code

5.1.1. Background

Linux, and the POSIX API in general, provides an interface for applications to interact with the dynamic loader at runtime to load libraries and functions during program execution. This API is primarily provided by two functions (and several variants of them) [137]:

1. `dlopen`, which may load a shared object not part of the program's current address space, and

```c
// Version a (linked with -lm)
#include <math.h>
#include <dlfcn.h>

int main(void) {
  cos(1.0);
  return 0;
}
```

(a) An ordinary call to the `cos` function.

```c
// Version b
#include <dlfcn.h>
typedef double (*fptr)(double);

int main(void) {
  / Load libm.so
  void *hdl = dlopen("libm.so", 0);

  // Get address of cos
  fptr func = dlsym(hdl, "cos");

  func(1.0);
  return 0;
}
```

(b) Calling `cos` dynamically using `dlsym`.

Figure 5.1: Example usages of `dlopen` and `dlsym`.

2. `dlsym`, which looks up the address of a code or data symbol (*i.e.,* a function or global variable) by name, either in a specific shared object or from the program's complete address space.

As opposed to the standard methods for specifying required shared objects and function usages in ELF metadata, `dlopen`/`dlsym` allows programmatic usage of shared libraries and functions *at runtime*. This enables applications to provide dynamic features like plugin frameworks, dynamic dispatch, or to test for libraries providing optional features. However, dynamically-loaded code introduces challenges for reconstructing the program's callgraph, as the functions loaded may only be known during execution.

Figure 5.1 shows an example program using the math function `cos` in a basic form (a) and rewritten using `dlopen`/`dlsym` (b) to look up the address of the function. In version (a), analysis of the disassembled program can determine that `cos` is called through a `call` instruction to the function's PLT. However, in (b), the address of the call instruction is returned by the call to `dlsym`—and based on `libm.so`'s location in the address space—so its address cannot be resolved statically using the same methods.

To handle these cases, `sysfilter` can take as input a user-provided "DL file" specifying a set of symbols and shared libraries that are loaded dynamically. These extra functions are added to the callgraph as additional starting points for the VCG and are always considered reachable for the construction of syscall sets. While this method provides a complete analysis, it requires using an out-of-band method to identify the set of functions and libraries loaded dynamically. This could be performed manually a developer or administrator with knowledge about how the program will execute, or generated from a runtime tracing tool. However, it remains an open question to what extent our techniques can identify dynamically-loaded symbols.

5.1.2. Resolving symbol names automatically

One opportunity to handle this is to use static value tracking, which `sysfilter` already uses to resolve syscall numbers (§ 3.2.3), to resolve the *string* argument to `dlsym` and map it to the corresponding function in the analysis scope. In Figure 5.1(b), the argument to `dlsym` (*i.e.,* the string "cos") is a constant and can therefore be read from the `.rodata` section of the ELF binary. For these cases, argument tracking shows promise for resolving the names of dynamically-loaded symbols, which can then be looked up in the program's analysis scope in a manner similar to `sysfilter`'s "DL files."

However, not all symbol or shared library names can be resolved to constant values. First, only constant string values can be resolved using static value tracking. The example in Figure 5.2a constructs the argument to `dlsym` dynamically in a buffer stored on the stack, rather than as constant data in the `.rodata` section, so its value cannot be resolved using the same methods. Moreover, the argument comes from variable `func_name`, which itself is an argument to the function `run_init`. While static tracking of certain values through memory or function calls may be feasible in some cases, we avoid this and use a strictly inter-procedural value tracking approach, as the alternative would increase the complexity of the tool significantly and potentially lead to path explosion.

Second, and more critically, even with highly complex data-flow analysis, many dynamic loads cannot be resolved statically using the binary alone. Figure 5.2b shows an example where the arguments to both `dlopen` and `dlsym` are parsed from a configuration file, which is a common behavior for applications with plugin support. Here, the library and symbol name are only known at runtime by parsing the configuration file. Thus, knowledge of the system configuration is required to determine which functions must be loaded. We note that this is a limitation of *any* static analysis tool, regardless of the techniques used, as the code to be loaded depends on the system configuration or program inputs.

Finally, even complete knowledge of the string values passed to `dlsym` may not be sufficient for constructing the a complete callgraph that contains all dynamically-loaded functions. Figure 5.2c shows a function `get_foo` that attempts to load function `foo_v2` and instead loads

```
#include <string.h>

typedef void (*fptr)(void);

void run_init(char *func_name)
{
  char buf[10];
  strncpy(buf, 10, "init_");
  strncpy(buf, 10, func_name);

  fptr func = dlsym(hdl, buf);

  func();
  return 0;
}
```

(a) Dynamically-constructed string

```
config cfg = parse("/etc/foo.conf");
hdl = dlopen(cfg.plugin_path, 0);
func = dlsym(hdl, cfg.plugin_name)
func();
```

(b) Symbol/library names from an external file

```
fptr get_foo(void) {
  func = dlsym(hdl, "foo_v2");
  if (func != NULL) {
    return func;
  }
  return dlsym(hdl, "foo_v1");
}
```

(c) Non-existant symbols

Figure 5.2: `dlopen`/`dlsym` with dynamic values.

function `foo_v1` if `foo_v2` is not present. Both function names (*i.e.*, `foo_v2` and `foo_v1`) are constant values and can be recovered, but this is no guarantee that the code for both functions is actually present on the system. In the case of `get_foo`, a functioning program only requires one `foo_*` function to be loaded successfully—e.g., `foo_v1` may be a legacy version of the function that is only present in very old libraries and not found on the system. Since some symbol loads may be expected to fail, this creates challenges for determining when all developer-intended dynamic function loads have been resolved. Handling these cases may require additional information about older symbols that can be ignored, or, if all dynamically-loaded libraries are known, a "best effort" approach may be used to ignore symbols that are not present in the current library version.

For these reasons, we conclude that while static value tracking my assist with resolving dynamically loaded symbols, *any* complete approach requires some knowledge of the program's runtime environment to construct the callgraph with certainty. To build our approach, we leverage our distribution-wide studies of programs in Debian `sid` to identify common usages of dynamic loading and determine requirements for system-dependent resolution.

5.1.3. `dlsym` usage in the wild

To characterize the extent of dynamically-loaded code, and to what extent symbol names can be resolved statically, we conducted further distribution-wide studies of ≈30K binaries in Debian `sid` using our `autopkg` framework (§ 4.2). We conducted our studies in parallel with our syscall

Library	**Usage type**	**Calls resolvable** **dlopen**	**dlsym**	**Calls**	**Binaries affected**
glibc	gconv	--	100%	95.8K (11%)	31.9K (100%)
glibc	Exception handling	--	100%	176.0K (21%)	31.9K (100%)
glibc	Name Service Switch (NSS)	--	23%	59.1K (7%)	18.4K (57%)
XCursor	Fallback symbol load	100%	100%	36.4K (4%)	3.5K (11%)
Kerberos	Plugin load	0%	0%	13.0K (2%)	4.3K (14%)
OpenGL	Dynamic dispatch	71%	99%	285.9K (33%)	4.3K (14%)
OpenSSL	Plugin load	0%	0%	13.2K (2%)	3.3K (10%)
libp11-kit	Plugin load	0%	100%	5.8K (<1%)	2.9K (9%)
libsasl	Plugin load	0%	0%	4.0K (<1%)	2.0K (6%)
GOMP	GCC OpenMP plugins	0%	100%	59.5K (7%)	1.6K (5%)
Other shared libraries		11%	66%	91.5K (11%)	10.5K (33%)
Application-specfic usages		70%	63%	16.6K (2%)	2.8K (9%)
All dlopen/dlsym calls		45%	89%	856.8K (100%)	31.9K (100%)

Table 5.1: Categories of `dlopen`/`dlsym` usage. For each category, we report the number of `dlopen` and `dlsym` calls for which argument tracking was able to resolve a constant value. glibc uses internal mechanisms to load libraries instead of `dlopen`, so its usage is not reported here—however, we expect that these cases do not resolve to constant values.

set analyses by extending `sysfilter` to report on usages of `dlopen` and `dlsym` for each binary and attempt to recover the library name (for `dlopen`) and symbol name (for `dlsym`) passed in their first argument. In our studies, we found that adding these additional static value tracking passes had negligible impact on analysis performance (cf. Figure 4.5, "Other Value Tracking").

Overall, we observe that static argument tracking was largely successful at recovering values, but that it is not sufficient for resolving symbol names on its own. In ≈30K binaries analyzed, we observed ≈851K `dlopen`/`dlsym` callsites (herein "DL callsites"), of which 45% of `dlopen` calls and 89% of `dlsym` calls could be resolved statically. Critically, in order to provide a *safe*, *complete* reconstruction of the callgraph *all* DL calls for a binary must be resolved, which was only true only ≈20% of binaries that made DL calls.[1]

However, we note common patterns in DL call usage that show promise for improving the analysis. Table 5.1 shows the prevalence of DL call usages by popular categories. A significant number of DL calls originate from shared libraries: only 2% of DL callsites, found in only 9% of the binaries, originated in the application itself. We manually inspected some of the most common shared library calls and matched them to well-known library APIs: over 80% of the DL callsites observed pertain to the ten cases listed, which comprise common system libraries. The top three cases (comprising 39% of all callsites) originate *inside glibc* to implement locale-specific character

[1]Excluding optional DL calls inside glibc for exception handling and gconv, which we discuss shortly.

set conversions (gconv), Name Service Switch (NSS), and dynamic lookups for exception handling information. The other cases correspond to prominent plugin and dispatch mechanisms in well-known libraries, such as Kerberos modules [138] and OpenGL's `glvnd` dynamic dispatch framework [139].

While static argument tracking alone is not sufficient for resolving all DL callsite arguments, our results demonstrate that a significant number of calls can be resolved with built-in handling for common library APIs that can identify and load required functions based on the library's respective system configuration. We therefore devoted our effort to identifying an effective paradigm for adding this functionality to `sysfilter`—and, by extension, our larger `libfcg` framework (§ 6.1)—to enable flexible, scalable development of handlers for common libraries. We term these handlers *resolvers*, which are pluggable modules that identify dynamically-loaded code for specific libraries.

As a case study, we explored how identify dynamically-loaded functions are used in glibc's NSS framework and extend `sysfilter` to automatically resolve these cases. We chose NSS as it is the most popular library we identified with significant failures and its usage can result in drastically different syscall sets. Our experience implementing NSS informed our design to realize a generic framework for adding further resolvers to `sysfilter` to support other libraries in the future.

5.1.4. Case study: handling GNU NSS

The Name Service Switch (NSS) subsystem [140] provides an interface for applications and system services to access user databases and name resolution services. NSS is used to resolve domain names, user and group names, and other service and directory information, and is thus essential for nearly all applications that use network and directory services. On Linux, NSS is part of glibc and can be used via standard API calls for translating names—e.g., `gethostbyname` to look up domain names, or `getpwent` to get information about a username.

The key principle of NSS is modularity. The main API provides a series of *databases*, which are accessed by programs using common glibc API functions. For example, the "passwd" and "shadow" databases contain user identity and password information, respectively, and can be accessed using the glibc functions `getpwent` and `getswent` (among others), respectively. The `hosts` database contains information about resolving machine hostnames and domain names (e.g., `localhost` or `google.com`). One function that can query the hosts database is `gethost-`

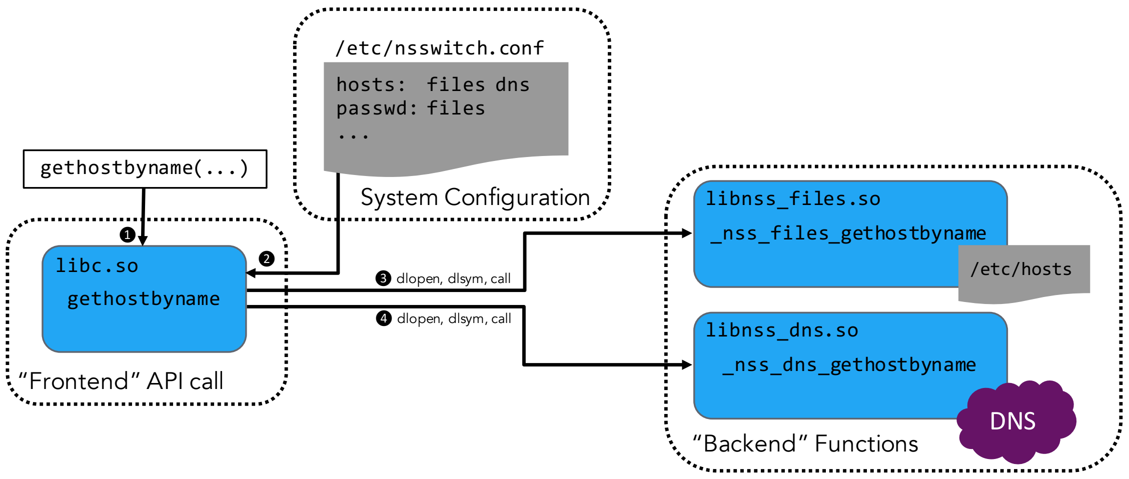


Figure 5.3: NSS resolution example.

`byname`, which translates domain names to network addresses (e.g., IP addresses). Databases are implemented by a series of *services*, which are shared libraries that implement a set of API calls to query the appropriate name information for each database. The file `/etc/nsswitch.conf` contains a list of services providing each database. This architecture allows developers to create custom service implementations to support additional functionality without extending glibc. Accordingly, NSS services have a wide range in complexity, from simple file lookups (like querying `/etc/passwd`) to rich, featureful modules for enterprise-level directory and authentication systems (e.g., NIS, LDAP, SASL, *etc.*).

Figure 5.3 illustrates the NSS resolution process for a call to `gethostbyname`, which uses the `hosts` database. When the function is called (❶), glibc parses the configuration file (❷) and dynamically loads (using `dlopen`) the service libraries listed for the `hosts` database, *i.e.*, the `files` and `dns` services and sequentially calls the appropriate backend function in each library. Using the configuration in Figure 5.3, glibc will first query the "files" service (❸) by loading `libnss_files.so` and then calling the function `_nss_files_gethostbyname`, which looks up the name in `/etc/hosts`. Depending on the return value, glibc may either return a result, or dynamically load and query the next service. In this case, if a result is not found in `/etc/hosts`, glibc will query the the `dns` service (❹), which will try to resolve the name using the DNS protocol.

To summarize, the NSS subsystem has three relevant interfaces: the "frontend" glibc API functions that use NSS features, the "backend" functions provided by service libraries, and the configuration file. When an NSS frontend function is called, glibc dynamically loads a set of backend functions from the service libraries using `dlopen`/`dlsym`, as determined by the configuration file,

`/etc/nsswitch.conf`. We find that this high-level view is shared among other plugin-based subsystems we examined, which guided our design in `sysfilter`.

5.1.5. Resolver Implementation

We extended `sysfilter` to automatically detect NSS usage and include the appropriate backend service functions in the callgraph. Our implementation over-approximates the set of backend functions that may be dynamically-loaded by considering the program's callgraph, the NSS configuration file, and a mapping between frontend and backend functions generated using an offline analysis. We designed our implementation as a generic, modular *resolver* which can be replicated to handle dynamically-loaded code in other libraries or plugin systems.

Offline function mapping The key component of the resolver is a mapping between frontend NSS functions, *i.e.,* glibc API functions that cause NSS code to be loaded dynamically, and the backend functions that are loaded from the NSS service libraries. We constructed this mapping in an offline manner using a combination of manual examination of glibc's source code and automatic analysis with `sysfilter` itself. We ran `sysfilter` on each exported function in glibc to identify which frontend functions interact with the NSS subsystem and load code dynamically using `dlsym`. This provided a list of the total set of glibc frontend functions that used NSS for investigation. By examining the NSS subsystem source code, we found that the names of the backend functions were dynamically crafted from constant strings that were passed to several internal glibc functions to perform NSS lookups. We used our static value tracking on these functions (examples include `__nss_library_load` and `__nss_hosts_lookup2`) to resolve these names manually, which was successful for most usages. We manually examined any callsites where value tracking failed to identify any other functions loaded and complete the mapping.

While we were able to leverage `sysfilter` to construct the function mapping, we emphasize that this is *not* a fully-automatic process. Rather, `sysfilter`'s callgraph construction and static value tracking aided our manual analysis by identifying the total set of functions to investigate and creating an initial mapping from constant string values. Expert oversight was required to determine *how* to apply argument tracking to glibc and to fill in any gaps in the analysis based on knowledge of the source code. While the full process requires developer oversight, we believe our analysis techniques, and other state-of-the-art analysis tools can significantly reduce the workload.

Online resolution After the analysis scope for the program has been constructed (§ 3.2.1), the NSS resolver examines the dynamic symbol table (`.dynsym`) for each shared object to find any

NSS frontend functions used in the program. Notably, the dynamic symbol table is present in all dynamically-linked shared objects, as this metadata is required to support dynamic linking. Once the set of frontend functions has been identified, the resolver determines which NSS service libraries may be loaded by parsing `/etc/nsswitch.conf`. Each library specified for the required services is then disassembled and added to the analysis scope. Using the function mapping, we find the set of backend functions matching each frontend function. For the example with `gethostbyname` in Figure 5.3, the resolver will load the shared objects corresponding to each library in the `hosts` database, (`libnss_files.so` and `libnss_dns.so`, respectively) and locate the matching backend functions for `gethostbyname`. The resulting list of backend functions represents a safe, tight overapproximation of the set of functions that glibc may load dynamically when `gethostbyname` is called.

The result of the resolver pass is a map of frontend API functions to the set of backend functions that may be loaded at runtime. During callgraph construction (§ 3.2.2), if a frontend function is considered reachable, its corresponding backend functions are added to the set of reachable functions as well. This ensures that the dynamically-loaded functions are added only if they are reachable, maintaining the tightest overapproximation.

To validate our NSS resolver, we used our resolver implementation in our correctness tests and distribution-wide studies described throughout Chapter 4 and Chapter 5. In each of our studies, we ran `sysfilter` on each binary using a standard NSS configuration from a default install of Debian `sid`. We believe that this configuration represents a reasonable baseline for a typical Debian system. We note that system configurations that use more featureful NSS libraries, such as network-backed directory services like LDAP, may result in different system call sets for binaries that use NSS. For example, applications that fetch user information may require system calls to access the network (e.g., `socket`) if the NSS configuration contains network-based user services.

Supporting other resolvers We implemented our NSS resolver as a case study to demonstrate the feasibility of building similar resolvers for other common libraries that load functions at runtime (§ 5.1.3). Figure 5.4 shows the impact of our NSS resolver, and potential extensions for other libraries, for resolving DL callsites across our dataset. Each point represents the proportion of binaries in our dataset that would have all of their `dlopen` and `dlsym` usages resolved—and thus a fully complete callgraph that includes dynamically-loaded code—given a certain makeup of resolvers. The x-axis denotes the combined set of resolvers for the top 10 libraries (Table 5.1), e.g., the point "Kerberos" represents the number of binaries that would be resolved if all DL callsites related to Kerberos were resolved, as well as those for XCursor, NSS, etc.

In the present version of `sysfilter`, which uses our NSS resolver, we are able to resolve dynamically-loaded strings to determine all dynamically-loaded code for ≈50% of the binaries in our dataset. (Again, we consider the callsites for `gcov` and glibc exception handling as resolved without any special handling, as these are strictly optional features, and the names loaded are fully known.) Adding resolvers for the top 10 libraries, could resolve all DL callsites for up to ≈60% of binaries. Beyond this, implementing resolvers for other shared libraries has further potential impact, up to ≈90% of our dataset, as shared libraries are responsible for 98% of DL callsites. We also note that the proportion of binaries covered does not change significantly with when adding naive, automatic, string-based resolution, suggesting that library-specific resolvers can provide the best impact.

Based on our implementation, we believe that building resolvers is a feasible solution for handling dynamically-loaded code in common libraries. The initial cost of building a resolver is high, as it requires an in-depth understanding of how each library may load code dynamically and where it may utilize information about the system (e.g., configuration files). However, this is a one-time and that can be aided by state of the art binary analysis tools to highlight cases for manual analysis. In Chapter 6, we discuss how we have extracted the relevant functionality from `sysfilter` as a generic framework to support similar analyses.

5.2. Partitioning the callgraph

Our current analysis techniques focus on identifying all reachable functions in the program by constructing a safe, tight representation of its FCG. In `sysfilter`, we have used this representation to identify the total set of system calls used by a program. In Chapters 3 and 4, we have shown that identifying reachable functions, and, by extension, syscall sets, is feasible for a large number of binaries, and effective for reducing application privileges. Following this, we explored the feasibility and effectiveness of *partitioning* the callgraph to allow for further privilege reductions at certain parts of the program.

In `sysfilter`, the system call set covers all possible execution paths in a program. While we have shown that this is effective for reducing overprivilege with respect to the full OS system call API, it follows that certain components of the program may be overprivileged *with respect to the program itself*. For example, a simple webserver may use the `bind` and `listen` syscalls at startup to open and listen on a serving port, and then spawn worker threads to handle client requests.

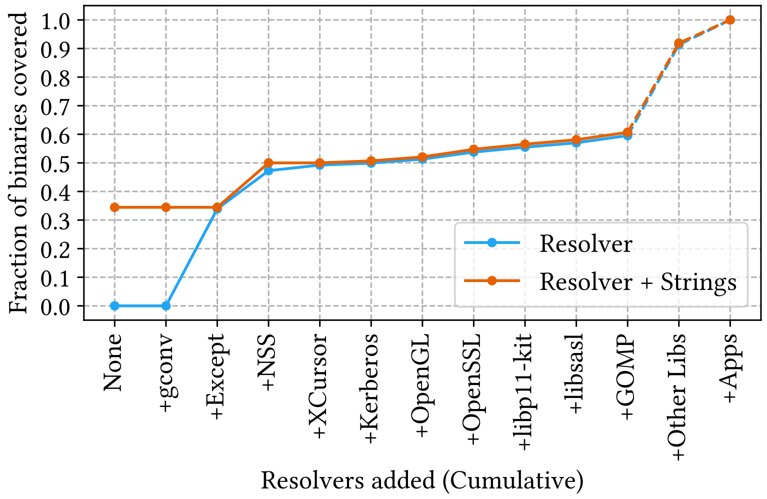


Figure 5.4: Theoretical impact of DL resolvers for common libraries. "Resolver" indicates the proportion of binaries that would have all `dlopen/dlsym` callsites resolved given the cumulative set of resolvers indicated on the x-axis. "Resolver + Strings" indicates the proportion of binaries resolved if string values for other DL callsites are also considered resolved—this represents the best-case scenario for an automatic attempt to resolve `dlopen/dlsym` strings not handled by a resolver.

Since the worker threads do not need to open new ports to merely serve requests, the worker threads are overprivileged as they have access to `bind` and `listen`, even though they are not required for operation. In this case, the overprivileged have a significant security impact, as the unnecessary access to `bind` and `listen` provides and adversary the capability to open ports for new connections, an extremely valuable primitive for escalating an attack.

In order to further improve the program's security posture in these cases, we aimed to use our callgraph construction techniques to identify partitions in program execution that resulted in safe, effective reductions in application privileges. Specifically, we conducted studies to determine the feasibility and security impact of *automatically* partitioning the callgraph at two well-known *transition points* in the program's execution:

1. At program startup, *i.e.*, before and after `main` is called
2. At creation of new threads with `pthread_create`

We select these transition points for our studies as they represent changes in program execution that are relatively simple to identify in automatic processing and apply to a wide range of binaries: nearly *all* binaries have an identifiable `main` function, and ≈53% of binaries in our dataset (*i.e.,* ≈17K programs) use `pthread_create`. In addition, these transition points represent stages of program execution where some runtime penalty is already required due to kernel involvement (*i.e.,* due to program initialization or thread creation, respectively), making them good opportunities for switching enforcement policies without introducing significant performance impact.

Similar to our analysis of dynamically-loaded code (§ 5.1), our approach is data-driven: we leverage our `autopkg` infrastructure to analyze two partitioning schemes on binaries in Debian `sid` to assess their effectiveness in reducing privileges in a large set of programs. In the remainder of this section, we discuss the mechanics of constructing each partitioning scheme, discuss challenges inherent to partitioning the callgraph, and discuss our findings based analysis with on our dataset. Overall, we focus our studies on assessing the feasibility of creating partitions at these transition points and their effectiveness at scale, with the goal of informing the development of future, partition-aware enforcement mechanisms.

Related work in partitioning Matching system call usage to phases of program execution has been widely studied for host-based intrusion detection, which we discuss in § 2.1.1. In terms of partitioning system call usage to reduce application privileges, and concurrently with our work, Ghavamnia et al. [13] perform *temporal specialization* of system call sets instrumenting transition points between the initialization and serving phases of common server applications. Their compiler-based approach relies on expert identification of the transition point between application and serving phase and uses source-level information to prune the callgraph for each partition. Our efforts focus on *automatically* identifying transition points a without requiring source code—we consider this a complementary analysis that can trade off partition specificity with scalability to a larger number of applications without requiring expert intervention.

5.2.1. Fixed partitions: before and after `main`

Our initial partitioning approach, and perhaps the most straightforward, involves separating the program's functionality during and after program initialization, *i.e.,* before and after *main* is called. From a developer's perspective, the execution of a standard C/C++ program begins at `main`. However, all programs run a number of initialization routines *prior* to calling `main` in order to set up the program. Accordingly, this initialization code requires system calls to function, and thus adds to the set of privileges required by the program.

Consider the most minimal C program (e.g., `int main(void) { return 0; }`), which makes no system calls from `main`. Running `sysfilter` on this program will result in a system call set with 40 syscalls, owing to code contained in the program's initialization routines, and other code considered reachable from them as determined by the callgraph extraction process (§ 3.2.2). Therefore, initialization routines represent an opportunity for partitioning programs to reduce privileges that may be required only by initialization routines.

Figure 5.5 shows a simplified view of the program startup process for dynamically-linked binaries compiled using glibc. Program initialization functions are typically divided among initialization code linked into each binary (usually from the `crt*.o` family of object files, which are provided by the toolchain) and setup routines built into the standard C library. Program execution begins at the entry point specified in the ELF binary, usually `_start`, which performs operations to set up the program stack and calls `__libc_start_main`, a function in `libc.so` that performs basic initialization tasks including: setting up global data structures (e.g., `.bss`, initializing thread storage, and other setup routines for the standard library itself. In addition, `__libc_start_main` calls the constructors for the program as specified in the `.init` and `.init_array` sections of the program ELF, as well as other constructors specified by shared libraries, and finally calls `main`. Once `main` returns, the program calls any destructor functions registered in the `.fini` and `.fini_array` sections of the binary and any shared libraries, runs other exit handlers, and, finally, terminates execution. A complete discussion of all initialization and finalization routines involved in this process is beyond the scope of this document—we refer interested readers to [141] for more details.

Creating the callgraphs To extract the program's system call usage before and after `main` is called, we generate the program's callgraph in two partitions, as shown in Figure 5.5. To construct the callgraph for each partition, we construct the vacuumed callgraph (VCG, § 3.2.2) as usual, but restrict the set of starting functions to the entry points for the partition. The first partition, $FCG_{\texttt{init}}$, is generated from the program's initialization routines, including `_start` and any constructors found in the binary and shared libraries.

For the second partition, $FCG_{\texttt{main+fini}}$, we use the program's `main` function and any destructor functions present in `.fini` and `.fini_array` as starting points for constructing the VCG. In our prototype implementation, this requires that the main application binary (but not necessarily its shared libraries) have symbol information in order to look up the address of the `main` by its name—1142 binaries (3.5% of our dataset) did not meet this requirement. Locating `main` without symbol information is also feasible without symbols by using value tracking to recover the ad-

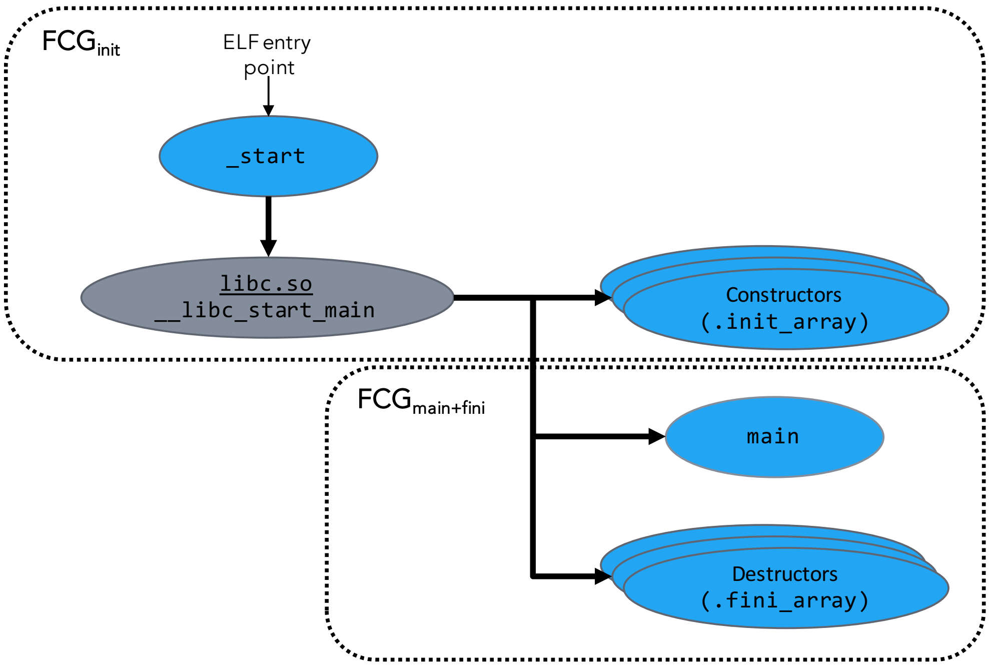


Figure 5.5: Partitioning before and after `main`. $FCG_{\texttt{init}}$ and $FCG_{\texttt{main+fini}}$ represent the scope of the partitions we identify for our analysis. Note that the elements of program initialization/finalization shown here is not a complete view of all functions executed, but represents the sets of functions considered as the starting points for constructing the callgraph for each partition.

dress passed to `__libc_start_main`; we leave this implementation to future work to support more binaries.

Handling address-taken (AT) functions At this stage, the resulting callgraphs represent a subset of the callgraph for the entire program, $FCG_{\texttt{all}}$, composed from a restricted set of starting functions. However, the partitioned callgraphs are inherently incomplete, as they may contain address-taken (AT) functions that were address-taken in other parts of the program. Our partitioned callgraphs must include these functions in order capture a safe, complete representation of the FCG.

Figure 5.6 shows an example program with some initialization code in a constructor `ctor`. For simplicity, we ignore other initialization code not shown in this listing, such as `_start`. To begin, we construct a naive representation the callgraph for both partitions, $FCG'_{\texttt{init}}$ and $FCG'_{\texttt{main+fini}}$ using the VCG construction process as described in the previous paragraph. From this step, the current (incomplete) representation of the set of reachable functions V for each partition can be

```c
typedef void (*fptr)(void);

void f6(void) { ... }
void f5(void) { ... }
void f4(void) { ... }
void f3(void) { ... }
void f2(void) { ... }
void f1(void) { f3(); }

fptr g_ptr = NULL;

__attribute__((constructor))
void ctor(void) {
  g_ptr = &f1;
  f4();

  fp = &f6;
  fp();
}

__attribute__((destructor))
void dtor(void) { ... }

void main(void)
{
  g_ptr();
  f5();

  return 0;
}
```

Figure 5.6: Example program with initialization routines. Functions `ctor` and `dtor` are registered as constructor and destructor functions, respectively, using the given compiler attributes.

written as follows:

$$\begin{aligned} V[FCG'_{\texttt{init}}] &= VCG(\{\texttt{ctor}\}) = \{\texttt{ctor}, \texttt{f1}, \texttt{f3}, \texttt{f4}, \texttt{f6}\} \\ V[FCG'_{\texttt{main+fini}}] &= VCG(\{\texttt{main}, \texttt{dtor}\}) = \{\texttt{main}, \texttt{f5}, \texttt{dtor}\} \end{aligned} \tag{5.2.1}$$

The constructor loads the address of `f1` into the global pointer `g_ptr`, which is then called by `main`. Thus, $FCG_{\texttt{main+fini}}$ is not complete, as it does not contain `f1`. This is because the VCG locates AT functions by identifying code pointers created as it traverses the callgraph (§ 3.2.2). Since `ctor` is not traversed in $FCG'_{\texttt{main+fini}}$, `f1` is not found.

To resolve this safely, for each partitioned callgraph we must also include the set of address-taken functions found for the VCG of the complete program. Recall that the VCG is composed as follows:

$$V[VCG] = (V[DCG] \cup V[ACG]) \tag{5.2.2}$$

where DCG is the callgraph created from all direct function calls, and ACG is a pruned representation of all reachable AT functions in the program. For any partition FCG_*, we can find the complete representation for its reachable functions as the union of the VCG for the partition and the program's ACG:

$$V[FCG_*] = (V[VCG_*] \cup V[ACG]) \tag{5.2.3}$$

In the example, ACG contains the pruned set of AT functions identified in Figure 5.6:

$$V[ACG] = \{\texttt{f1}, \texttt{f6}\} \tag{5.2.4}$$

In practice, `main` is itself an address-taken function, as it is called from `__libc_start_main`. We exclude `main` from this set as a special case, since it should only be called once, at program start, in any reasonable program. Thus, the ACG can be added to each partition to construct a complete representation of the callgraph for each part of the program, $FCG_{\texttt{main+fini}}$ and $FCG_{\texttt{init}}$:

$$\begin{aligned} V[FCG_{\texttt{init}}] &= VCG(\{\texttt{ctor}\}) \cup V[ACG] = \{\texttt{ctor}, \texttt{f1}, \texttt{f3}, \texttt{f4}, \texttt{f6}\} \\ V[FCG_{\texttt{main+fini}}] &= VCG(\{\texttt{main}, \texttt{dtor}\}) \cup V[ACG] = \{\texttt{main}, \texttt{f1}, \texttt{f5}, \texttt{f6}, \texttt{dtor}\} \end{aligned} \tag{5.2.5}$$

The addition of the ACG from the entire program represents an safe, but significant overapproximation of the callgraph for each partition. In the example, `f6` is only called by `ctor`, yet it remains as part of $FCG_{\texttt{main+fini}}$. In our tests, this overapproximation resulted in very similar syscall sets compared to the original program. We discuss our results and potential mechanisms for reducing the number of AT functions added in § 5.2.2.

Partitioned syscall filters Once the partitioned callgraphs have been constructed, we determine the system call sets for each partition. Critically, syscall policies added after the start of program execution can only *add* restrictions to previous policies. This is a necessary security requirement to prevent a compromised, restricted process from modifying the policy to increase its privileges. Our system call filters are enforced in kernel mode using `seccomp-BPF`, which allows *stackable* filters and only permits system calls if they are allowed by all attached filters.

To determine system call sets from our partitioned callgraphs, we must compose the sets such that any transitions yield a more restrictive set. For fixed partitions before and after `main` is called, this results in two system call sets (S): an initial set $S_{\texttt{all}}$ representing the total syscall set for the program, and $FCG_{\texttt{main+fini}}$ representing the syscall set after `main` is invoked. These can be composed from the partitioned callgraphs as follows:

$$\begin{aligned} S_{\texttt{all}} &= S[FCG_{\texttt{init}} \cup FCG_{\texttt{main+fini}}] \\ S_{\texttt{main}} &= S[FCG_{\texttt{main+fini}}] \end{aligned} \tag{5.2.6}$$

Thus, a program with a system call set partitioned in this way would install a system call filter on startup (cf. § 3.2.4), and then add another system call filter before calling `main` to *drop* any syscalls that are no longer required.

Partitioning on thread creation We also explored the effectiveness of partitioning the program's callgraph at the start of new threads, with the goal of further restricting program privileges at multiple points in the program. Since a new thread of execution may indicate a separate component of a program's execution, this partitioning scheme offers the potential for identifying differing system call sets between threads, such as the webserver example discussed at the start of § 5.2. The process of constructing the partitioned callgraph for each thread function is mostly similar to partitioning before and after `main`, with the added challenge of automatically identifying the thread functions.

Figure 5.7 shows an example program that creates three threads. To identify the functions that are spawned as new threads, we leverage `sysfilter`'s value tracking analysis to identify function addresses in a similar manner to locating syscall numbers (§ 3.2.3) and `dlopen/dlsym` arguments (§ 5.1.2). Specifically, we configure `sysfilter` to track instances of `pthread_create` and attempt to fetch the value of the third argument, which denotes the "start routine" for the new thread. If a constant value is recovered, we search the program's analysis scope to find a function with a matching address.

Candidates for thread partitioning In our studies on Debian `sid`, we found that ≈53% of the ≈30K binaries in our study used `pthread_create` to create at least one thread. As with our other uses of value tracking, the data-flow analysis may fail to recover a value if the argument to `pthread_create` is not a constant address, e.g., if the value is computed dynamically, passed as a function argument, or loaded from the stack or heap. We observed ≈48K `pthread_create` callsites in our dataset, ≈54% of which could be resolved successfully.

In contrast to our analysis for dynamically-loaded code, we do not require all thread routine

```c
#include <pthread.h>
typedef void (*fptr)(void);

pthread_t pt1, pt2, pt3;

void f4(void) { ... }
void f3(void) { ... }
void f2(void) { ... }
void f1(void) { ... }

fptr g_fptr = NULL;

void *t1(void *arg) {
  g_fptr();
  f4();
}

void t2(void *arg) {
  fptr fp = (fptr)arg;
  fp();

  pthread_create(&pt3, NULL, &t3, (void*)0);
}

void *t3(void *arg) {
  f3();
  return NULL;
}

void main(void)
{
  // ... Initialize pt1, pt2, pt3 ...
  g_fptr = &f1;

  pthread_create(&pt1, NULL, &t1, (void *)0);
  pthread_create(&pt2, NULL, &t2, (void *)&f2);

  return 0;
}
```

Figure 5.7: Thread partitioning example.

functions to be known in order to apply thread partitioning. Instead, this optimization could be applied to any thread for which its address (and the address of any threads spawned by its children, as we discuss later) are known. We observed ≈11K binaries in which at least one thread routine address could be resolved, with ≈7.8K binaries (or ≈70% of binaries that create threads) meeting this requirement for all child threads. Across the entire dataset, this indicates that 24% of programs could support thread partitioning for at least one thread function.

Constructing the callgraph For each thread function, we create the FCG in a similar procedure to the partition for `main`. We construct the VCG using the thread routine as a starting function, and add ACG from the entire program to cover all AT functions that may be called from the thread. Thread routines may utilize AT functions from global data, e.g., `t1` in Figure 5.7, may also use function pointers passed to the thread during creation via the fourth argument (`arg`) to `pthread_create`, e.g., `t2`. While it is feasible to apply our value tracking approach to this argument as well, we found that this argument is almost *never* constant and thus offers limited opportunity for recovering function pointers.

For example, based on the program in Figure 5.7, we can compute the ACG and the FCG for thread routines `t1` and `t2` as follows:

$$V[ACG] = \{\texttt{f1}, \texttt{f2}, \texttt{t1}, \texttt{t2}, \texttt{t3}\} \tag{5.2.7}$$

$$V[FCG_{\texttt{t1}}] = VCG(\{\texttt{t1}\}) \cup ACG = \{\texttt{t1}, \texttt{f4}\} \cup ACG = \{\texttt{f1}, \texttt{f2}, \texttt{f3}, \texttt{f4}, \texttt{t1}, \texttt{t2}, \texttt{t3}\} \tag{5.2.8}$$

$$V[FCG_{\texttt{t2}}] = VCG(\{\texttt{t2}\}) \cup ACG = \quad \{\texttt{t1}\} \quad \cup ACG = \{\texttt{f1}, \texttt{f2}, \texttt{f3}, \texttt{f4}, \texttt{t1}, \texttt{t2}, \texttt{t3}\} \tag{5.2.9}$$

Similar to the process for partitioning before and after `main`, the AT functions added from the ACG represents a safe, but significant, overapproximation of the call targets for each indirect call in `t1` and `t2`, resulting in both threads having the same set reachable functions. Indeed, since *all* thread routines are inherently address-taken functions in order to be passed to `pthread_-create`, all thread routines, and all functions reachable from them via direct or indirect calls, will be present in the ACG. For example, `f4` is reachable from `t2` via the direct call in `t1`, which is present in the ACG.

Thread routine `t3` offers a rare opportunity to reduce this overapproximation. Since `t3`'s function set ($\{\texttt{t3}, \texttt{f3}\}$) contains no indirect call instructions, it cannot call *any* address-taken function, and thus the entire ACG can be pruned from its callgraph:

$$V[FCG_{\texttt{t3}}] = VCG(\{\texttt{t3}\}) = \{\texttt{t3}, \texttt{f3}\} \tag{5.2.10}$$

While this optimization appears to show promise for restricting the set of AT functions, it occurs rarely in practice. In our dataset, we only observed a very small number of binaries (<20) with thread functions that contained zero indirect calls.

Constructing syscall policies for thread partitions is similar to partitioning before and after *main*. For each thread function, we construct a syscall set based on the set of reachable functions for the thread routine and any thread routines it calls. In Figure 5.7, `t2` spawns thread `t3`, so the resulting system call sets for each thread are as follows:

$$\begin{aligned} S_{\texttt{t1}} =& S[FCG_{\texttt{t1}}] \\ S_{\texttt{t1}} =& S[FCG_{\texttt{t2}} \cup FCG_{\texttt{t3}}] \\ S_{\texttt{t3}} =& S[FCG_{\texttt{t3}}] \end{aligned} \tag{5.2.11}$$

In order to construct syscall policies for thread partitions, *all* thread routines spawned from a given thread must be known. If any of the child thread routines cannot be recovered via argument tracking, as it could spawn a thread routine with a larger system call set. In our dataset, we found that ≈70% of binaries with any single thread routine resolvable met this requirement. Given that our current AT function handling strategy involves adding all AT functions to all partition callgraphs, requiring all thread routine functions to be resolved via argument tracking is not required, as each thread routing is inherently address-taken at some point in the program. However, we continue to report results using this child thread requirement, in order to better inform studies that use more restrictive methods for pruning AT functions in each thread partition.

5.2.2. Partitioning effectiveness in Debian `sid`

We evaluated the effectiveness of each partitioning scheme as part of our large-scale studies on Debian `sid`. For each binary considered in our study, identified the proportion of binaries that could support each partitioning method and extracted the relevant information to obtain the system call sets for each partition type. Thus, for each binary, we collected the system call sets for the full-program VCG (as in a typical `sysfilter` run), $FCG_{\texttt{init}}$, $FCG_{\texttt{main+fini}}$, the ACG, and the FCG for each thread function identified by value tracking. For this study, we created combined system call sets for binaries meeting the conditions of each partition type in postprocessing rather than as part of `sysfilter` itself, which permitted flexibility in our analysis after collecting information about each binary. Once a viable partitioning scheme is identified, this component can be easily integrated into `sysfilter`'s existing codebase.

Table 5.2 compares the applicability of each partitioning scheme and overall effectiveness for reducing system call sets. Partitioning after `main` can be applied to all binaries, but offered little effectiveness for reducing privileges: reducing the system call set at `main` only resulted in smaller system call sets for 8% of binaries in our dataset. Thread-based partitioning is applicable to fewer binaries: we observed ≈11K binaries in which at least one thread routine could be resolved, ≈70%

	After `main`	**Thread-based (>1 child routine)**	**Thread-based (all child routines)**
Binaries considered	31933 (100%)	8768 (51%)	7838 (46%)
Binaries with reduced syscall sets	2466 (8%)	8768 (51%)	7838 (46%)
Median syscalls removed	0	5	5

Table 5.2: Comparison of partitioning effectiveness. "Binaries considered" reports the number of binaries that supported partitioning using the given approach. "Binaries with reduced syscall sets" lists binaries for which at least one system call could be removed in any partition. All percentages are relative to the total number of binaries in the study (31933).

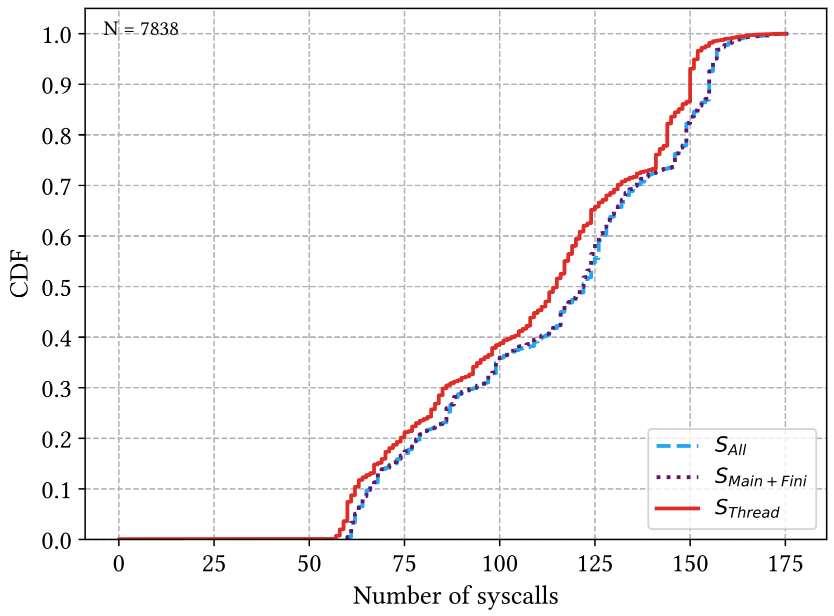


Figure 5.8: Distribution of system call sets with partitioning. Note that the system call set distributions for S_{All} and $S_{Main+Fini}$ are virtually identical. For S_{Thread}, the thread with the largest system call set is reported.

of which had all child threads resolved and were thus considered for partitioning. We note that additional engineering effort for resolving thread function pointers using data-flow tracking may increase the total set of closer to the true number of binaries using `pthread_create`.

Across the 7838 binaries supporting thread-based partitioning, we identified 13149 thread routines, with 90% of binaries using $\leq$3 identifiable threads. Table 5.5a lists the most common system calls that could be removed in at least one thread partition per binary. The most common syscall removed was `set_tid_address`, a benign syscall rarely used outside glibc initialization, which could be removed from nearly all binaries. The next most common syscalls relate to the `timerfd` and `epoll` interfaces, which could be removed from nearly half of the binaries in this dataset,

Syscall	Binaries Removed	% of Usages Removed
`sendmmsg`	970	6.20
`sched_getaffinity`	644	22.24
`statfs`	464	5.36
`inotify_add_watch`	330	2.61
`inotify_init1`	330	2.62
`timerfd_create`	280	2.14
`timerfd_settime`	280	2.14
`epoll_create1`	278	2.14
`epoll_wait`	264	2.02
`getegid`	236	1.47

(a) After `main` (31933 binaries)

Syscall	Binaries Removed	% of Usages Removed
`set_tid_address`	7837	99.99
`timerfd_create`	3972	76.30
`epoll_create1`	3933	77.27
`timerfd_settime`	3856	74.00
`epoll_wait`	3774	73.21
`sendmmsg`	2201	40.44
`inotify_init1`	2157	44.56
`inotify_add_watch`	2134	43.90
`getpeername`	1196	20.85
`sched_setaffinity`	642	8.19

(a) Thread-based (7838 binaries)

Table 5.5: Most common removed syscalls by partition type. For each system call, we report the number of binaries where the given syscall could be removed from at least one partition and the percentage of binaries using this system call for which it could be removed.

which is ≈70% of the binaries using these system calls. Based on the set of "security-critical" system calls compiled by Ghavamnia et al. [13] (comprising 17 syscalls such as `bind`, `execve`, and `mprotect`), only ≈10% of thread partitions (≈1.3K threads) removed a critical syscall.

Figure 5.8 compares the distribution of system call set sizes across all binaries for each partition type for the binaries supporting both schemes. Partitioning after `main` had a negligible impact on the size of system call policies. For binaries supporting thread-based partitioning, we observed that the syscall set size in individual partitions decreased by ≈5% (5 syscalls) at the median, and ≈9% at the 90th percentile (10 syscalls), when compared to their respective binary. We did not observe significant variations in system call sets *between* threads in the same binary—this is expected, since each thread shares the same set of AT functions, which itself includes the set of all thread functions.

Overall, we believe that these small differences in system call sets across partitions is primarily due to the large overapproximation for AT functions. Indeed, the ACG alone accounts for most of the system call set across all applications. Figure 5.9 shows the proportion of a program's system call set that could be recovered from the ACG alone. Over 50% of binaries have >96% of their system call set reachable from the just the ACG, creating an upper bound on the effectiveness of our partitioning approaches. Therefore, despite the limited effectiveness we observed, we do not claim that these partitioning schemes provide little security value. Rather, we show that our analysis techniques can feasibly identify these types of partitions, and do so automatically across

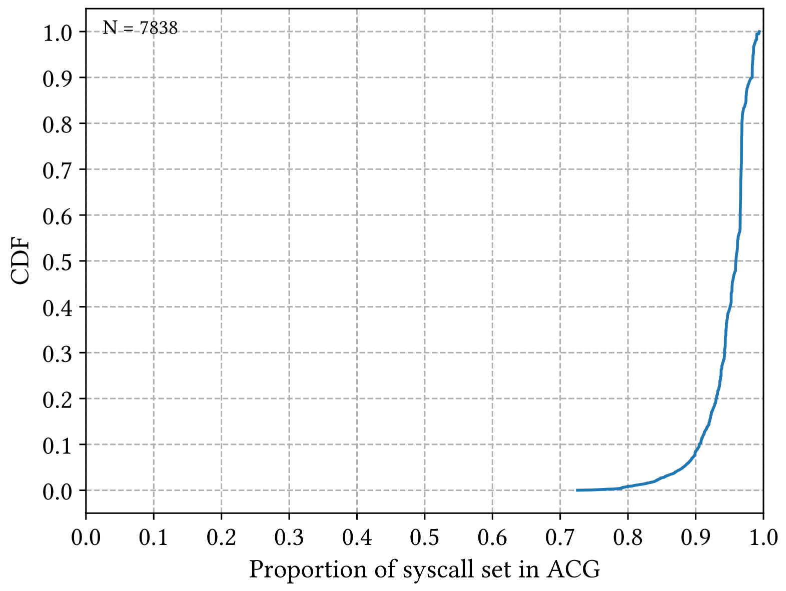


Figure 5.9: Comparison of ACG vs. VCG syscall set sizes. This distribution shows the proportion of a binary's system call set `sysfilter` finds when searching *only* AT functions in the program (*i.e.*, from the ACG).

a large set of binaries. Our results demonstrate the need for further methods to prune the set of AT functions matched to indirect call targets in each partition (and across the program as a whole), which can improve the effectiveness of future partitioning schemes.

Accurately determining the set of AT functions for indirect calls, or points-to analysis, is a fundamental and well-studied problem in program analysis. In general, precisely identifying the set of values of a function pointer using static analysis alone is undecidable [142, 143], both at the binary-level, or using source code for memory-unsafe languages. Research efforts in this area have spanned nearly three decades, and focus on various approximation strategies for different program analysis contexts, including system call-based intrusion detection [9, 47, 144], control-flow integrity (CFI) [38, 39, 40, 145, 146], attack-surface reduction [13, 111], among others. `sysfilter`'s approach (§ 3.2.2) uses relocation information to provide the complete set of AT functions in the program (which is then pruned further in the VCG), thus creating a callgraph that is sound (*i.e.*, it never excludes a potential AT function), but can provide a significant overapproximation of the set of indirect call targets. This approach is consistent with contemporary practices for determining indirect targets solely from the program binary [13, 147, 148]. Other works have demonstrated further pruning when more information about AT functions and indirect callsites is available, including using source code to prune AT functions by type or number of arguments [13, 40, 149], or using runtime tracing to identify used targets [150, 151]. Pruning based purely on binary analysis, as would be most ideal for our tools, is much less common. TypeArmor [152] uses dataflow analysis to infer the number of function call arguments to refine the set of indirect targets to improve CFI strategies for disrupting COOP attacks, but it remains an open question as to whether this analysis can be applied for callgraph reconstruction. We leave these questions for future work—should such binary-only pruning methods become available, integrating them into our partitioning framework can improve the effectiveness of our partitioning schemes, and our callgraph construction as a whole.

Chapter 6

Generic tools for reducing overprivilege

6.1. `libfcg` framework

While `sysfilter`'s primary purpose is to identify a program's system call usage, many of our analysis techniques are not specialized to this goal. Our analysis constructs a representation of the program's callgraph by overapproximating the set of reachable functions from the program's binary and shared libraries. Beyond system call filtering, this representation has applications in other program analysis domains, such as attack surface reduction and software debloating. Accordingly, we realized the generic components of our tools as `libfcg`, a framework for identifying reachable code in a program binary, in order to facilitate development of new security tools using our analysis techniques.

Figure 6.1 shows the architecture of `libfcg`. `libfcg` provides a library, built atop the Egalito framework, which includes our implementation to compose the vacuumed callgraph (VCG) and find the set of reachable functions. In addition, we include our implementations for our modules to resolve dynamic loading (§ 5.1.5) and static argument tracking (§ 3.2.3), which can be used to improve the precision of the generated callgraph and extract features about the program (*i.e.*, syscall numbers, function arguments), respectively. Conceptually, `sysfilter` is an extension from these generic components, using the VCG construction process to generate a representation of the callgraph, and then using static argument and value tracking to recover system call numbers.

As we continue to develop our tools, our goal is to make `libfcg` a generic library of tools that can support development of new security tools and techniques involving static program analysis. Indeed, during our offline analysis to discover NSS usages in glibc (§ 5.1.4), we used `libfcg`'s argument tracking on several internal glibc functions, which helped us to identify how symbol

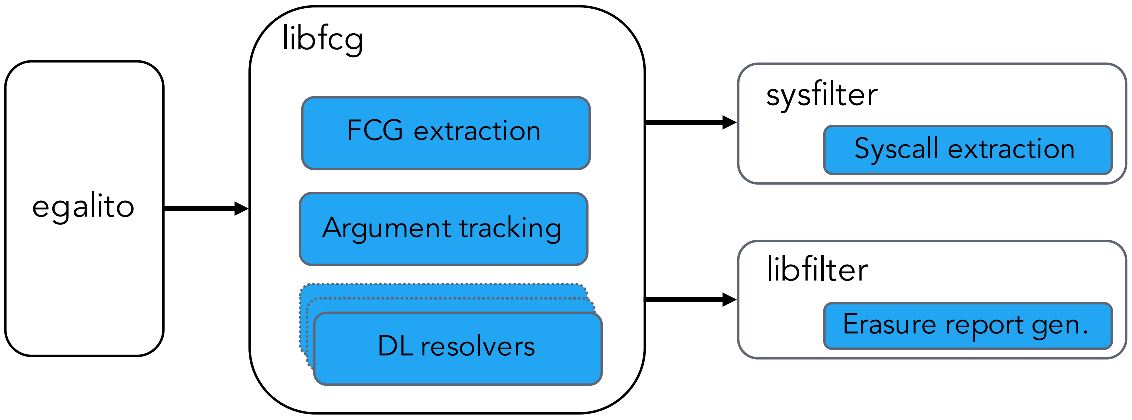


Figure 6.1: `libfcg` architecture.

names were passed to dynamic load functions. This demonstrates a simple example for how some or all of `libfcg`'s tools can be used for more generic program analysis tasks. As we continue to use and develop these tools, we intend to continue developing `libfcg` as a robust, extensible framework.

To further explore how well our techniques could support security mechanisms outside system call filtering, we used `libfcg` to develop `libfilter`, a tool to reduce overprivilege in terms of library code usage. In the remainder of this chapter, we discuss our development of `libfilter` and leverage our `autopkg` framework for deploying our tools at scale to assess its scalability and report findings on the prevalence of code bloat in Debian `sid`.

6.2. `libfilter`: Debloating binary shared libraries

Using `libfcg`, we developed `libfilter`, a tool to identify unused functions in binaries and their shared libraries. `libfilter` is a *software debloating* tool: it identifies unused library functions based on a program or set of programs so that unused functions can be removed. By removing unused functions, `libfilter` can generic *specialized* libraries that contain only the library functions necessary for an application, reducing the "ammunition" available to an attacker for launching a code reuse attack.

`libfilter` is based on contemporary efforts in software debloating research. Concurrently and independently from our work with `sysfilter` and `libfcg`, Agadakos et al. developed Nibbler [69], a static analysis tool to debloat shared libraries, which employs similar callgraph construction methods. After the publication of Nibbler, we extended this work together by implementing `libfilter`, a more scalable debloating tool based on `libfcg`. Specifically, we implemented Nibbler's FCG construction component using `libfcg` to and identify shared library functions that are unused and thus can be removed. Since `libfilter`'s design goals are extremely similar to Nibbler, we omit the design details and focus on our prototype implementation, how it uses `libfcg`, and how it improves upon Nibbler's initial prototype.

Prototype implementation Building on `libfcg`'s capabilities to over-approximate the FCG of a target program, `libfilter`'s implementation is relatively straightforward. Like `libfcg`, the input to `libfilter` is an ELF file containing a main application binary, from which `libfcg` builds an analysis scope from the binary and its required shared libraries (§ 3.2.1) and parses the program from its entry points to construct the VCG (§ 3.2.2). The VCG represents an over-approximation of all code, in both the main binary and its shared libraries, that can be executed by the program for any input. From this point, `libfilter`'s task is straightforward: enumerate the set of all shared library functions to find those that are *not* present in the VCG, which represents the set of unused functions. For each application binary, `libfilter` outputs a list of all shared library functions, its location information in the program (*i.e.,* its offset into its respective shared ELF object) and a boolean flag indicating if it can be erased. This information can be passed to a compatible debloating tool, such as Nibbler or an Egalito-based rewriting tool, to generate *thinned* versions of the shared libraries containing only the required functions.

Our prototype implementation of `libfilter` comprises only 313 LOC atop our existing implementation for `libfcg`. Like `sysfilter`, our implementation for `libfilter` is organized as an Egalito "pass" atop our existing codebase to construct the VCG. Combined with our Egalito-based implementation of `libfcg`, for ≈1.4 KLOC total, this is a substantial reduction compared to the ≈7 KLOC of Python code required by Nibbler. In addition, our use of Egalito's improved jump table analysis methods provide improved precision compared to Nibbler's original implementation, which was based on the `objdump` utility [90].

Evaluation: Correctness We evaluated `libfilter` in terms of correctness in a similar manner to `sysfilter` (§ 3.4). In our preliminary evaluation, we erased 109 binaries from three projects: GNU coreutils, Nginx webserver, and GNU m4. For each project, we erased all binaries using `libfilter` and tested them using the projects' unit test suites, comprising 1143 unit tests in total.

In our preliminary tests, we found that `libfilter` can successfully debloat real-world problems, with some exceptions related to dynamic loading of symbols and libraries. Similar to our

Binary	Symbol loaded dynamically	Test File
`cp`	`_xstat`	`tests/cp/nfs-removal-race.sh`
`df`	`fopen`	`tests/df/no-mtab-status.sh` `tests/df/skip-duplicates.sh`
`hostid`	`__libc_readline_unlocked`	`tests/misc/help-version.sh`

`ls`	`getxattr`
	`lgetxattr`
	`print_call_count`
		`tests/ls/getxattr-speedup.sh`

`pinky`	`__libc_readline_unlocked`	`tests/misc/help-version.sh`
`rm`	`readdir`	`tests/rm/rm-readdir-fail.sh`
`uniq`	`setvbuf`	`tests/misc/stdbuf.sh`
`true`	`setvbuf`	`tests/misc/help-version.sh` `tests/misc/stdbuf.sh`

Table 6.1: Dynamically-loaded symbols used by `libfilter` correctness tests. All binaries listed are from GNU coreutils. Note that many feature pertain to optional features, like extended attributes or readline support, which may suggest why these symbols are loaded dynamically.

`sysfilter` evaluation, we exclude tests involving dynamic loading to support arbitrary scripting languages, such as Perl support in Nginx. We found 11 tests that required us to manually include symbols that were loaded dynamically (e.g., using `dlsym`), shown in Table 6.1. In these cases, our current FCG construction approaches do not automatically find these functions and mark them as used as the address of the function is determined at runtime by `dlsym`—thus, it never appears in the statically-constructed call graph. This type of dynamic function usage is problematic in any static analysis approach, e.g., [69]. In § 5.1, we discuss strategies for automatically handling dynamically-loaded code in certain libraries, reducing the amount of manual intervention required in these cases.

6.2.1. Library bloat in the wild

In our preliminary `libfilter` study, we processed 30631 binaries across 8641 packages, of which 34537 (≈91.7%) could be analyzed successfully (the rest 8.3% corresponds to binaries with missing symbols, non-C/C++ code, *etc.*). Across the set of all the successfully-analyzed binaries, we observed 5295 unique dynamic shared libraries. The median time to process each binary is around 20s, with 90% of binaries completing in under 200s—this is a *substantial* improvement compared to Nibbler's, Python-based prototype, which required more than 60s per binary [69].

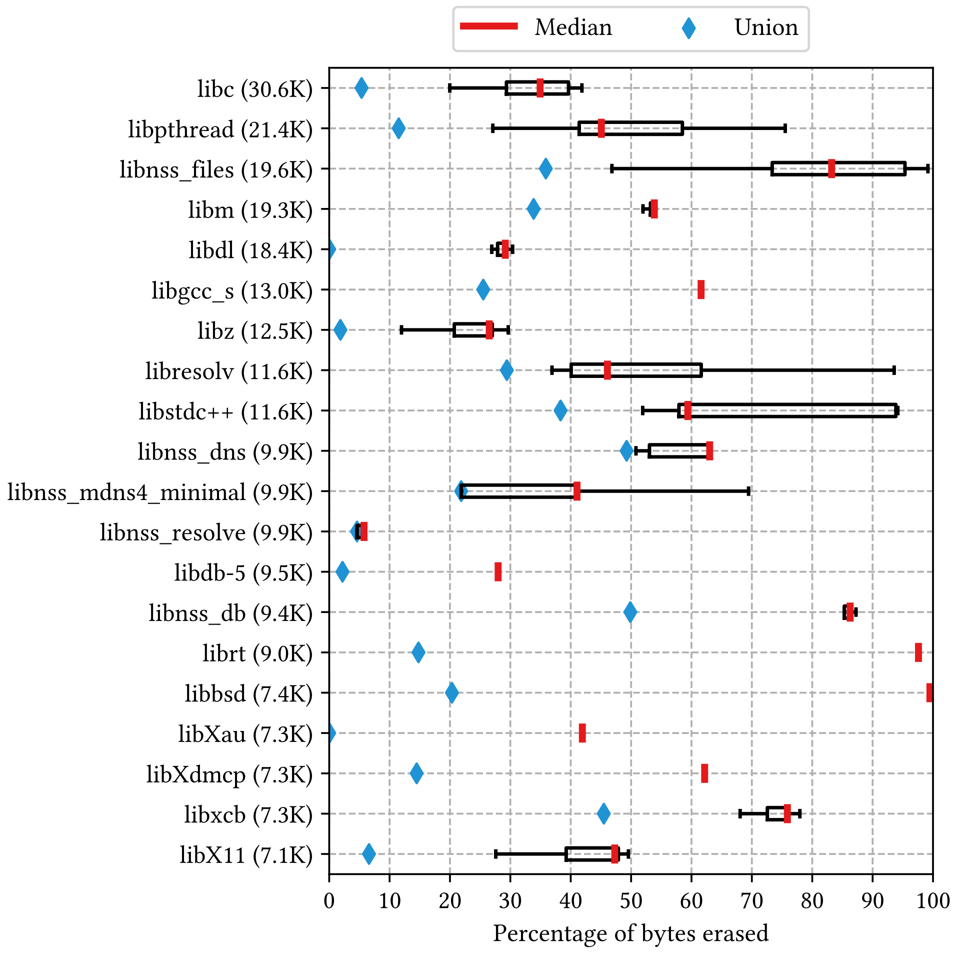


Figure 6.2: Unused code in the 20 popular libraries in Debian `sid`. Numbers in parenthesis indicate the number of binaries (out of the 30631 analyzed) using the respective library.

Bloat in popular libraries For each binary, `libfilter` reports the set of libraries used by the application, the total set of functions in each library, and the set of functions that are unused and can be removed. We aggregate these results across all the ≈30K considered binaries to *determine* and *characterize* code bloat at large (*i.e.,* across the whole Debian distribution).

Figure 6.2 shows the distribution of the amount of code removed (x-axis) for the top 20 libraries observed (y-axis), as a percentage of their total size. (Shared libraries are ranked based on the number of packages they link-with.) With one exception, all the binaries we analyzed (*i.e.,* 30631) link-with `libc`; the exception is the binary `mtcp_restart`, an executable with no shared library dependencies.

For `libc`, we observe that the median amount of code removed, per binary, is ≈35%, with most binaries individually not requiring 20%–42% of its code. The next most popular libraries are

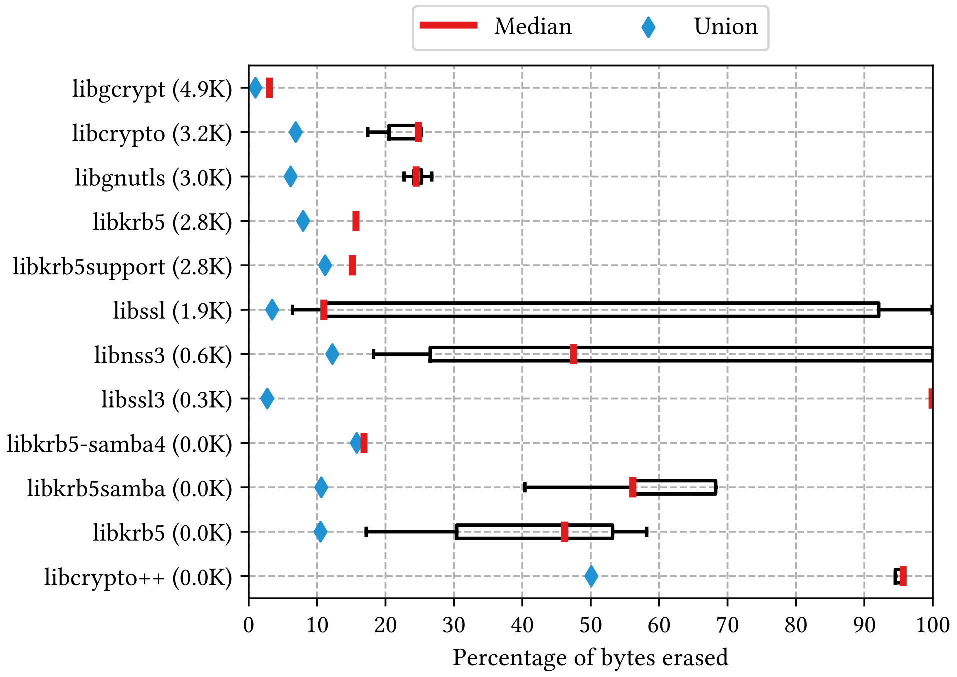


Figure 6.3: Unused code in cryptographic libraries in Debian `sid`. Libraries used by $\leq$ 20 packages are excluded. Numbers in parenthesis indicate the number of binaries (out of the 30631 analyzed) using the respective library.

`libpthread` and `libnss_files`, used by ≈20K and ≈19K binaries, respectively, demonstrating much wider distributions (*i.e.*, 27%–76% and 47%–99%), similarly to `libresolv` and `libstd++` (37%–99%, 52%–94%). In contrast, `libnss_resolve` (≈9K% binaries) has a very tight distribution with a median of only ≈5% of code removed, owing to the relatively small number of functions it exports and their tight coupling (*i.e.*, all variations of `gethostbyname` and `gethostbyaddr`). `libm`, `libdl`, `librt`, and `libbsd` also exhibit tight distributions, with a median of ≈54%, ≈29%, ≈97%, and ≈99% of code removed, respectively.

Beyond per-binary metrics, we can gain further insights by considering the *union* of all functions required across all binaries in the distribution, which identifies code not used by (almost) *any* binary in the system. Accordingly, the union amount of code to remove is always lower than the median reported in Figure 6.2. For `libc`, we observe that ≈5% of its code is unused across all binaries. As another example, the popular NSS [140] libraries `libnss_dns` and `libnss_db` have nearly 50% unused code across all binaries; likewise, `libxcb` and `libstd++` exhibit ≈46% and ≈38% unused code across all binaries.

Lastly, Figure 6.3 and Figure 6.4 illustrate the effect of `libfilter` on popular cryptographic,

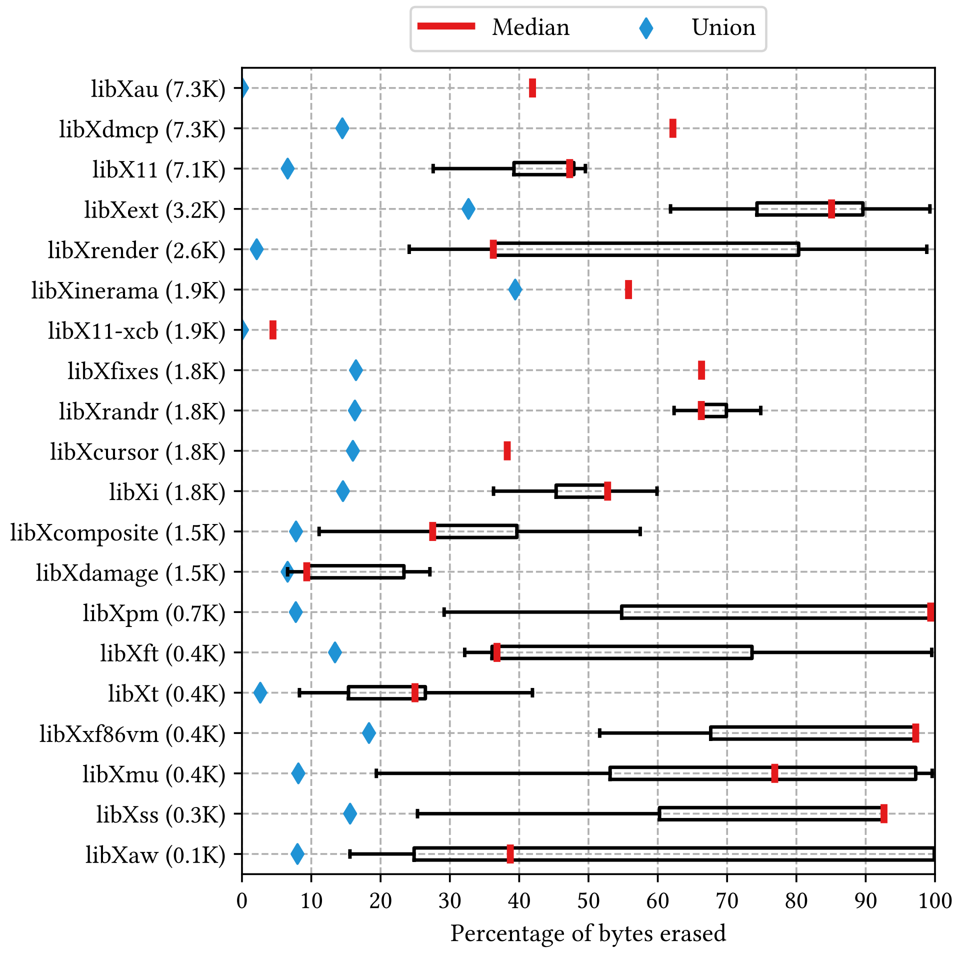


Figure 6.4: Unused code in X11 libraries in Debian sid. Libraries used by $\leq$ 20 packages are excluded. Numbers in parenthesis indicate the number of binaries (out of the 30631 analyzed) using the respective library.

and X11, libraries, across the whole Debian distribution.

6.2.2. Container set debloating

To assess the amount of code bloat in commonly-distributed sets of Linux binaries, we used `libfilter` to examine packages from popular container images published in the Docker Hub repository [136]. Rather than analyzing binaries in each image directly, we leverage our existing dataset and infrastructure for processing Debian `sid` packages to provide an equivalent analysis. Specifically, we used the container images to identify *sets* of binaries to consider from our Debian `sid` dataset. By examining the union of library functions removed, across all binaries in each set, we can we can quantify the total set of unused code across all libraries in the published image.

			Bytes		Functions	
Image	**#Bin**	**#Lib**	**Total**	**Safe to Erase (%)**	**Total**	**Safe to Erase (%)**
haproxy	293	62	14.9MB	3.5MB (23%)	46.8K	17K (36%)
influxdb	309	83	16.4MB	3.3MB (20%)	52.2K	16.8K (32%)
mariadb	394	84	17.1MB	3.7MB (22%)	53.2K	18.4K (35%)
memcached	292	61	13.4MB	3.3MB (25%)	39.6K	14.6K (37%)
nginx	294	59	13.1MB	3.2MB (24%)	38.5K	14K (36%)
postgres	363	83	20.6MB	4.7MB (23%)	69.1K	23.8K (34%)
redis	295	68	15.5MB	3.6MB (23%)	48K	17.4K (36%)
spiped	296	60	14.5MB	3.5MB (24%)	44.9K	17K (38%)
ubuntu	310	61	13.2MB	3.2MB (24%)	39.1K	14.3K (36%)
Mean			15.4MB	3.6MB (23%)	48K	17026 (36%)

Table 6.2: Erasure for container image sets (using packages from Debian `sid`).

We selected 9 official Docker images for popular C/C++ applications, each with >10M downloads, on Docker Hub. To simplify the process of translating packages in the container's base distribution to Debian package names, we selected container images that were built from distributions using the `apt` package manager. For each container image, we mapped each binary to its providing package using `dpkg`, and then looked up the equivalent `sid` package in our dataset. Across all images, we considered a total of 255 unique packages. Mappings for all but 11 packages could be resolved automatically—we manually resolved the remainder, which differed only by version numbers (e.g., `mariadb-server-10.5` vs. `mariadb-server-10.4`). While this mapping between Linux distributions does not provide an exact representation of the unused code included in the container images, it does provide a *tight* approximation of code bloat across a commonly-installed set of packages in popular Docker images.

Table 6.2 shows the union amount of code removed across all libraries in each container image. Every image used roughly 300 binaries and less than 100 shared libraries. Across all images, we observe that an average of 23.2% of code (in terms of bytes) is unused. All images exhibited roughly the same amount of bloat, with image-wide union values within 4% of each other. The majority of this bloat is due to common system libraries used in each image, rather than application-specific libraries—we leave the analysis of comparing ancillary utilities and binaries required for the application's use-case as a future study. Interestingly, in 7 out of the 9 images, the library with the most code removed (>94%) was `libunistring` [153], a common library for handling Unicode. Conversely, all images utilized one library which had no code removed, *i.e.*, `libdebconfclient` [154], which is used by `dpkg`. Overall, our results demonstrate opportunities for reducing code bloat using `libfilter` in commonly-used container images.

Chapter 7

Conclusions and future work

As software evolves and becomes even more complex—and as libraries and system APIs expand their capabilities to match them—overprivilege becomes an increasing threat to system security. In order to defend against this threat, it becomes important to understand the prevalence of overprivilege across the wide range of deployed software and take actions to reduce it.

This thesis has investigated challenges in using static, binary-only analysis to develop a low-cost, effective method for reducing overprivilege in type- and memory-unsafe programs. Since overprivilege is widespread in fundamental OS interfaces, it affects nearly all programs that interact with the operating system. In this thesis, we have explored techniques for identifying and reducing overprivilege in programs *without* access to source code or dynamic tracing in order to develop robust tools to analyze the wide variety of complex problems that make up modern software ecosystems.

As a result, this thesis has developed tools to automatically characterize and reduce overprivilege by restricting programs to necessary features only, thus improving the security posture of systems by enforcing the principle of least privilege in a precise, effective, and scalable manner. We have demonstrated set of low-cost tools to identify overprivilege and restrict access to OS features in two domains: system call API usage and shared library code access. In doing so, we have used our tools to assess overprivilege a wide variety of programs, spanning an entire Linux distribution, to characterize overprivilege *in situ*, as well as demonstrate and our tools' scalability and effectiveness. In addition, we present a framework to extend and further apply our tools in other security domains.

7.1. Future work

Software is constantly evolving. As applications, libraries, and APIs continue to grow, new features and use cases will create more areas of overprivilege, with new requirements on how to restrict application privileges to improve security. In addition, as binary analysis techniques and enforcement mechanisms evolve, it will create new opportunities for deploying our current techniques to a wider range of programs and improving the precision and effectiveness of our analysis. In this section, we briefly outline future directions for extending and improving our techniques.

7.1.1. Refining indirect call targets

Our callgraph construction process (§ 3.2.2) provides an safe, tight overapproximation of the set of reachable functions in a program. The primary challenge in reconstructing the program's control flow is points-to analysis, *i.e.*, determining the set of targets for each indirect call in the program. This is fundamental and well-studied problem in program analysis—we discuss these challenges in § 5.2.2.

The approach used by `libfcg`—and thus by `sysfilter` and `libfilter`—is consistent with contemporary efforts for determining the complete set of address-taken (AT) functions used by indirect calls from the program binary alone. Improving points to-analysis analysis is an ongoing area of research, fueled by program analysis efforts in other security domains, such as CFI. Future developments in this area that could allow further pruning of AT functions for indirect callsites could significantly improve precision of our callgraph overapproximation, especially for partitions of the callgraph in the same program. Address-taken functions (*i.e.*, the total set of possible indirect call targets) make up a significant portion of any program's callgraph. In terms of system call sets, AT functions cover 95% of the syscall set for over half of binaries in our distribution-wide studies (§ 4.3.3). As binary analysis techniques evolve, we recommend further study of how new developments in points-to analysis can be integrated into our tools to improve the precision of our callgraph construction process.

7.1.2. Extending the scope of privilege reductions

Our tools have demonstrated effective privilege reductions in both system call API usage and library code access by identifying a program's required features, so that unused components can be restricted. A natural future direction is to explore how our analysis tools, available as an extensible framework with `libfcg` can identify required functionality in other OS interfaces, such as Linux capabilities [155], filesystem access, or usage of other kernel interfaces like `netlink` [156]. In addition, there is further opportunity to improve the precision of our existing syscall/library restrictions by identifying key values of system call arguments, such as flags passed to `socket` or `ioctl` to identify broad classes of functionality being requested. Identifying uses of other interfaces can benefit from our existing callgraph construction methods, as well as our static value tracking implementation, but will require additional effort to determine how specific API usages can be resolved using static analysis. A similar, but further-reaching, dimension is investigating how our tools may privileges may be from programs in other languages, potentially leveraging our existing binary analysis techniques to explore how runtimes for higher-level languages may use OS features.

7.1.3. Flexible system call policy enforcement

Current semantics for enforcing system call policies (such as `seccomp-BPF`) place restrictions on how policies are inherited by new processes or partitions within the same process (e.g., between threads). In broad terms, syscall policies added on top of existing policies can only *add* restrictions to previous policies. This is a necessary security requirement to prevent a compromised, restricted process from modifying its own policy to increase its privileges. However, this requirement creates difficulty for programs that create processes with a larger syscall set than its parent. We discuss these challenges in detail in § 3.2.4. To provide one motivating example, `sysfilter` is not well-suited for enforcing programs like shells, which inherently spawn arbitrary processes with diverse system call sets.

While these requirements for syscall filter composition are paramount for maintaining the security of arbitrary programs, we note an opportunity for allowing more flexibility when executing *trusted* applications. In an environment where the operating system could verify that a child process is a known, trusted program with its own system call policy, a more flexible enforcement mechanism could allow the child process to *switch* system call policies, rather than combining them, allowing it to use its full range of functionality. While this kind of enforcement model offers promising opportunities for applying system call filters to a wider range of programs, it would require a framework for verifying trusted processes. Development of trusted execution environments is an ongoing area of work in both research and industry—as these systems evolve, critical environments that use them could benefit from system call set restrictions using these more flexible policies.

7.1.4. Continued `libfcg` development

We have developed `libfcg` as an extensible framework to support new security tools using our callgraph construction and analysis techniques. As new tools evolve, we recommend incorporating new techniques and lessons learned into `libfcg` to benefit further security tools. As one example, `libfcg`'s DL resolver plugins (§ 5.1.5) can be extended to support new libraries over time as required by future analyses, improving the library's ability to handle dynamically-loaded code without user intervention. In addition, further development of `libfcg`'s static value tracking could improve its ability to recover constant arguments in more complex cases. Overall, improvements to `libfcg`'s core features, whether simply engineering improvements, or new analysis techniques, have the potential to improve precision, effectiveness, and scalability for a wider range of tools, contributing to security improvements across multiple domains.

Bibliography

[1] G. J. Holzmann, “Code inflation,” *IEEE Software*, no. 2, pp. 10–13, 2015.

[2] N. Hardy, “The Confused Deputy: (or why capabilities might have been invented),” *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[3] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[4] Linux Programmer’s Manual, “Syscalls – Linux system calls.” [Online]. Available: http://man7.org/linux/man-pages/man2/syscalls.2.html

[5] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos, “Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path,” in *USENIX Annual Technical Conference (ATC)*, 2017, pp. 1–13.

[6] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, “kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse,” in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 420–436.

[7] ——, “Kernel protection against just-in-time code reuse,” *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 1, pp. 1–28, 2019.

[8] Debian packages, “Package: Libc6.” [Online]. Available: https://packages.debian.org/buster/libc6

[9] D. Wagner and R. Dean, “Intrusion detection via static analysis,” in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2000, pp. 156–168.

[10] J. T. Giffin, S. Jha, and B. P. Miller, “Detecting manipulated remote call streams,” in *USENIX Security Symposium*, 2002, pp. 61–79.

[11] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security (TISSEC)*, pp. 61–93, 2006.

[12] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, 2008.

[13] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1749–1766. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[14] N. Provos, "Improving host security with system call policies," in *USENIX Security Symposium (SEC)*, 2003, pp. 257–272.

[15] V. Rothberg, "Generate SECCOMP Profiles for Containers Using Podman and eBPF," Oct. 2019. [Online]. Available: https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html

[16] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 869–886.

[17] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-Driven Software Debloating," in *Proceedings of the 12th European Workshop on Systems Security*. ACM, 2019, p. 9.

[18] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: Application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 329–339.

[19] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[20] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.

[21] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 380–394.

[22] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *USENIX Security Symposium (SEC)*, 2019, pp. 1733–1750.

[23] Debian Linux, "Statistics | Debian Sources." [Online]. Available: https://sources.debian.org/stats/

[24] Linux Programmer's Manual, "Syscall—indirect system call." [Online]. Available: http://man7.org/linux/man-pages/man2/syscall.2.html

[25] "Ringing in a new asynchronous I/O API [LWN.net]." [Online]. Available: https://lwn.net/Articles/776703/

[26] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security Symposium (SEC)*, 2012, pp. 459–474.

[27] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *USENIX Security Symposium (SEC)*, 2014, pp. 957–972.

[28] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 420–436.

[29] V. P. Kemerlis, "Protecting commodity operating systems through strong kernel isolation," Ph.D. dissertation, Columbia University, 2015.

[30] "NVD - CVE-2019-13272." [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2019-13272

[31] Solar Designer, "Getting around non-executable stack (and fix)." [Online]. Available: https://seclists.org/bugtraq/1997/Aug/63

[32] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[33] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[34] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 575–589.

[35] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 745–762.

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 969–986.

[37] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1868–1882.

[38] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[39] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[40] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1821–1835. [Online]. Available: https://doi.org/10.1145/3372297.3417867

[41] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 367–382.

[42] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for unix processes,” in *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 1996, pp. 120–128.

[43] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[44] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, “Exploiting execution context for the detection of anomalous system calls,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 1–20.

[45] A. Chaturvedi, S. Bhatkar, and R. Sekar, “Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments,” p. 19.

[46] D. Wagner and R. Dean, “Intrusion detection via static analysis,” in *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, May 2001, pp. 156–168.

[47] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, “Formalizing sensitivity in static analysis for intrusion detection,” in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, May 2004, pp. 194–208.

[48] J. T. Giffin, S. Jha, and B. P. Miller, “Automated discovery of mimicry attacks,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2006, pp. 41–60.

[49] Linux Programmer’s Manual, “Seccomp—operate on Secure Computing state of the process.” [Online]. Available: http://man7.org/linux/man-pages/man2/seccomp.2.html

[50] J. Corbet, “BPF: The universal in-kernel virtual machine.” [Online]. Available: https://lwn.net/Articles/599755/

[51] S. Dhillon, “[net-next v3 0/2] eBPF seccomp filters,” Mon Feb 26 07:26:54 UTC 2018. [Online]. Available: https://lists.linuxfoundation.org/pipermail/containers/2018-February/038571.html

[52] “Mozilla wiki: Security/Sandbox/Seccomp.” [Online]. Available: https://wiki.mozilla.org/Security/Sandbox/Seccomp

[53] "A safer playground for your Linux and Chrome OS renderers," Nov. 2012. [Online]. Available: https://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html

[54] "OpenSSH 6.0 release notes." [Online]. Available: https://www.openssh.com/txt/release-6.0

[55] Docker, "Docker." [Online]. Available: https://www.docker.com/

[56] Podman, "Podman." [Online]. Available: https://podman.io/

[57] Docker Documentation, "Seccomp security profiles for Docker," Mar. 2020. [Online]. Available: https://docs.docker.com/engine/security/seccomp/

[58] T. Garfinkel, B. Pfaff, M. Rosenblum *et al.*, "Ostia: A delegating architecture for secure system call interposition." in *Network and Distributed System Security Symposium (NDSS)*, 2004.

[59] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, "Tailored Application-specific System Call Tables," Pennsylvania State University, Tech. Rep., 2014.

[60] DockerSlim, "DockerSlim - Lean and mean Docker containers. Smaller, faster, more secure and frictionless!" [Online]. Available: https://dockersl.im/

[61] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining Sandboxes for Linux Containers," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 92–102.

[62] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-Phase Execution of Application Containers," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017, pp. 230–251.

[63] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[64] C.-T. Lee, Z.-W. Rong, and J.-M. Lin, "Linux kernel customization for embedded systems by using call graph approach," in *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003.* IEEE, 2003, pp. 689–692.

[65] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "System-wide compaction and specialization of the Linux kernel," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 95–104.

[66] Debian, "Debian Installer / mklibs." [Online]. Available: https://salsa.debian.org/installer-team/mklibs/blob/master/src/mklibs

[67] G. Vigna and C. Kruegel, "BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings*. Springer, 2019, p. 482.

[68] C. Mulliner and M. Neugschwandtner, "Breaking Payloads with Runtime Code Stripping and Image Freezing," 2015.

[69] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 70–83.

[70] J. Kinder and H. Veith, "Precise static analysis of untrusted driver binaries," in *Formal Methods in Computer Aided Design*. IEEE, 2010, pp. 43–50.

[71] N. Davidsson, A. Pawlowski, and T. Holz, "Towards Automated Application-Specific Software Stacks," *arXiv:1907.01933 [cs]*, Jul. 2019. [Online]. Available: http://arxiv.org/abs/1907.01933

[72] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[73] G. S. Misherghi and Z. S. HDD, "Hierarchical delta debugging," PhD Thesis, Citeseer, 2007.

[74] J. Geffner, "VENOM: Virtualized environment neglected operations manipulation." [Online]. Available: http://venom.crowdstrike.com

[75] H. Zhao, Y. Zhang, K. Yang, and T. Kim, "Breaking turtles all the way down: An exploitation chain to break out of VMware ESXi," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

[76] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX Winter Conference*, 1993, pp. 259–270.

[77] The Linux Kernel, "Seccomp BPF (SECure COMPuting with filters)." [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

[78] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 48–62.

[79] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for c," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.

[80] ——, "CETS: Compiler enforced temporal safety for c," in *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2010, pp. 31–40.

[81] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and C++ programs," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 1–28, 2012.

[82] Common Weakness Enumeration, "CWE-123: Write-what-where condition." [Online]. Available: https://cwe.mitre.org/data/definitions/123.html

[83] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 574–588.

[84] B. Spengler, "PaX: The guaranteed end of arbitrary code execution," in *G-Con2*, 2003.

[85] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 276–291.

[86] Oracle Solaris, Linker and Libraries Guide, "Position-independent code." [Online]. Available: https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html

[87] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security Symposium (SEC)*, 2016, pp. 583–600.

[88] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 84–96.

[89] G. Corona, "The ELF file format." [Online]. Available: https://www.gabriel.urdhr.fr/2015/09/28/elf-file-format/

[90] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 133–147.

[91] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 128–142.

[92] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 227–242.

[93] Generic Part Linux Standard Base Core Specification, "Exception frames." [Online]. Available: https://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

[94] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[95] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *USENIX Security Symposium (SEC)*, 2019, pp. 1805–1821.

[96] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 147–163.

[97] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 941–951.

[98] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 461–477.

[99] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 584–598.

[100] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, 2011, pp. 522–536.

[101] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *USENIX Security Symposium (SEC)*, 2014, pp. 845–860.

[102] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 177–189.

[103] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *USENIX Annual Technical Conference (ATC)*, 2003, pp. 211–224.

[104] Linux Programmer's Manual, "Ld.so, ld-linux.so—dynamic linker/loader." [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[105] G. L. Steele Jr., "Debunking the "Expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The ultimate GOTO," in *ACM National Conference*, 1977, pp. 153–162.

[106] E. Bendersky, "Position independent code (PIC) in shared libraries." [Online]. Available: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

[107] G. Corona, "ELF loading and dynamic linking." [Online]. Available: https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/#library-resolution

[108] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "VTPin: Practical VTable hijacking protection for binaries," in *Annual Computer Security Applications Conference (ACSAC)*, 2016, pp. 448–459.

[109] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[110] P. Liang and M. Naik, "Scaling abstraction refinement via pruning," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 590–601.

[111] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated System Call Filtering for Commodity Software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 459–474. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/demarinis

[112] B. Dawson, "Symbols on linux update: Fedora fixes." [Online]. Available: https://randomascii.wordpress.com/2013/03/05/symbols-on-linux-update-fedora-fixes/

[113] D. Wiki, "Using symbols files." [Online]. Available: https://wiki.debian.org/UsingSymbolsFiles

[114] U. Wiki, "Debug symbol packages." [Online]. Available: https://wiki.ubuntu.com/Debug%20Symbol%20Packages

[115] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[116] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-Agnostic Binary Recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 133–147.

[117] Intel, "System v application binary interface," 2013. [Online]. Available: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[118] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2006.

[119] Linux Programmer's Manual, "Bpf – perform a command on an extended BPF map or program." [Online]. Available: http://man7.org/linux/man-pages/man2/bpf.2.html

[120] NixOS, "Patchelf – A small utility to modify the dynamic linker and RPATH of ELF executables." [Online]. Available: https://github.com/NixOS/patchelf

[121] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Network and Distributed System Security Symposium (NDSS)*, 2003, pp. 163–176.

[122] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[123] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 47–60.

[124] S. Moore, C. Dimoulas, D. King, and S. Chong, "SHILL: A secure shell scripting language," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 183–199.

[125] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 179–194.

[126] Debian, "The unstable distribution ("sid")." [Online]. Available: https://www.debian.org/releases/sid/

[127] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[128] Google Project Zero, "Speculative execution, variant 4: Speculative store bypass." [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[129] "Wrk – a HTTP benchmarking tool." [Online]. Available: https://github.com/wg/wrk

[130] Redis Labs, "NoSQL Redis and Memcache traffic generation and benchmarking tool." [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[131] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A study of modern Linux API usage and compatibility: What to support when you're supporting," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 16.

[132] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017, pp. 65–70.

[133] Debian Manpages, "Debootstrap." [Online]. Available: https://manpages.debian.org/stretch/debootstrap/debootstrap.8.en.html

[134] "Python APT Library." [Online]. Available: https://apt-team.pages.debian.net/python-apt/library/index.html

[135] gRPC Authors, "gRPC." [Online]. Available: https://grpc.io/

[136] Docker, "Docker hub." [Online]. Available: https://hub.docker.com

[137] Linux Programmer's Manual, "Dlopen - open and close a shared object." [Online]. Available: https://man7.org/linux/man-pages/man3/dlopen.3.html

[138] "For plugin module developers — MIT Kerberos Documentation." [Online]. Available: https://web.mit.edu/kerberos/www/krb5-latest/doc/plugindev/index.html

[139] "Libglvnd: The GL Vendor-Neutral Dispatch library," NVIDIA Corporation, Sep. 2021. [Online]. Available: https://github.com/NVIDIA/libglvnd

[140] The GNU C Library, "System databases and name service switch." [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html

[141] "Linux x86 Program Start Up." [Online]. Available: http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

[142] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 54–61.

[143] L. O. Andersen, "Program analysis and specialization for the C programming language," PhD Thesis, Citeseer, 1994.

[144] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient Context-Sensitive Intrusion Detection." in *NDSS*, 2004.

[145] M. Ghaffarinia and K. W. Hamlen, "Binary Control-Flow Trimming," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. London, United Kingdom: Association for Computing Machinery, Nov. 2019, pp. 1009–1022. [Online]. Available: https://doi.org/10.1145/3319535.3345665

[146] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." in *NDSS*, vol. 26, 2012, pp. 27–40.

[147] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000, pp. 323–337.

[148] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 559–573.

[149] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[150] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[151] C. Porter, G. Mururu, P. Barua, and S. Pande, "BlankIt library debloating: Getting what you want instead of cutting what you don't," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 164–180. [Online]. Available: https://doi.org/10.1145/3385412.3386017

[152] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[153] Free Software Foundation, “Libunistring.” [Online]. Available: https://www.gnu.org/software/libunistring/

[154] Debian, “Package: Cdebconf.” [Online]. Available: https://packages.debian.org/sid/cdebconf

[155] Linux Programmer’s Manual, “Capabilities—overview of Linux capabilities.” [Online]. Available: http://man7.org/linux/man-pages/man7/capabilities.7.html

[156] ——, “Netlink - netlink macros.” [Online]. Available: https://man7.org/linux/man-pages/man7/netlink.7.html