ATTACK SURFACE REDUCTION IN CONTEMPORARY OPERATING SYSTEMS VIA PRACTICAL KERNEL DEBLOATING

BY

HSUAN-CHI KUO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Associate Professor Sibin Mohan, Chair
Professor Ravishankar Iyer
Professor Vikram Adve
Assistant Professor Tianyin Xu
Assistant Professor Daniel Williams, Virginia Tech

ABSTRACT

This dissertation aims to address the problem of bloat in operating system (OS) kernels. It explores the problem by postulating that *modern operating systems have multiple sources of bloat — from code to runtime overheads — that increases attack surfaces and negatively affects performance: reduction of such bloat ("debloating"), in a practical way, can make operating systems more robust and efficient.* To validate this hypothesis, the work is divided into the following three groups:

1. Evaluate the tradeoffs between kernel debloating and kernel redesign (*e.g.*, unikernels)
2. Study kernel debloating techniques and validate their effectiveness
3. Develop a practical and effective kernel debloating framework

Previous work and this dissertation show that only a small part of a kernel is used by most applications. The redundant parts introduce performance regression (*e.g.*, prolonged boot time and higher memory footprint) and enlarge attack surfaces (*e.g.*, vulnerabilities due to software bugs). A study on using kernel debloating to reduce attack surfaces is conducted and shows that debloating is effective in reducing the redundancy as well as the kernel size (*e.g.*, more than 80% for most cloud applications). The study also identifies the limitations that prevent the practical usage of a kernel debloating including the amount of manual efforts and instability of produced kernels. This dissertation demonstrates the indispensable benefits of commodity OS kernel debloating by studying other specialization techniques (*e.g.*, unikernels). Unlike unikernels that only runs a small subset of Linux applications, the debloated Linux kernel not only runs every Linux application (full POSIX support) but also outperforms unikernels in various dimensions (*e.g.*, boot time, image size, memory footprint and application performance). Motivated by the benefits of kernel debloating, this dissertation explores debloating techniques by building a kernel orchestration framework (MultiK) and kernel profiling tools (DKut and SKut). The experiment results confirm that applications use only a small part of the kernel (*e.g.*, 93% of the kernel can be reduced for a web server). The results also show that aggressive and intrusive kernel debloating leads instability and cause kernel crashes, therefore, hindering its practical adoption. Based on the lessons learned, this dissertation further introduces an advanced and practical kernel debloating framework (Cozart) which debloats kernels automatically and generates stable kernels. I use Cozart as a vehicle to study how to make debloating more practical. I

share these insights and my experiences to shed light on addressing the limitations of kernel debloating in future research and development efforts. Finally, I go beyond the traditional definition of debloating and present KFuse that optimizes kernel extensions and reduce inefficiency.

to my wife, my family and those who have ever helped me in this wonderful journey.

ACKNOWLEDGEMENTS

This Ph.D. degree would have never happened without the help of many exceptional professors, mentors, collaborators and friends. I would first like to express my gratitude toward my fantastic advisor, Sibin. He admitted me to the Ph.D. program at University of Illinois Urbana-Champaign(UIUC), marked the beginning of this surreal journey and provided unlimited support to my study. I would also like to thank my doctoral committee, Professors Ravishankar Iyer, Vikram Adve, Daniel Williams and Tianyin Xu, for their insightful comments and feedback, which significantly improved the quality of this work.

I want to express my special appreciation to Tianyin Xu, who played the role of a professor and a friend at the same time. As a teacher, he treated me as his own student, providing guidance on being a great researcher. As a friend, he was always there to help, and it always felt great to hear from him.

I want to express my special thanks to Dan Williams, who was initially my mentor during my internship at IBM Research and became my long-term collaborator. Dan is always a nice person and a brilliant researcher. His contribution to this thesis is invaluable.

I would like to extend my gratitude to my collaborators and mentors, Xinyang Ge, Weidong Cui, Ricardo Koller, Jianyan Chen, Akshith Gunasekaran, Yeongjin Jang, Rakesh B Bobba, David Lie, Jesse Walker, Kai-Hsun Chen and Yicheng Lu for contributing to this thesis. The days we worked together were terrific. In addition, I would like to thank my labmates, Ashish Kashinath, Kyo Hyun Kim, Chaitra Niddodi and Bin-Chou Kao, for making this journey not lonely.

Last but not least, I would like to thank my parents and family for their love and support. Their unconditional support accompanied and comforted me throughout the days and nights in this strange country. In particular, my wife has been giving her unconditional love and commitment to supporting my study. My Ph.D. is not possible without you. I sincerely thank you for being with me throughout the ups and downs.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The complexity of general-purpose operating systems (OSes) has been increasing relentlessly. For example, the Linux kernel, Figure 1.1 shows the lines of code in the Linux kernel for different versions. The initial release of Linux has only thousand of lines of code, and Linux 4.19 already has more than 20 millions lines of code, and the number is still growing rapidly.

The introduced complexity and code is to support a variety of users from individuals to industries. Different software feature requests and hardware devices are emerging so the OS kernels have to catch up with new software features (*e.g.*, file systems, security mechanisms and optimizations), hardware supports (*e.g.*, peripheral device drivers) to various types of hardware drivers.

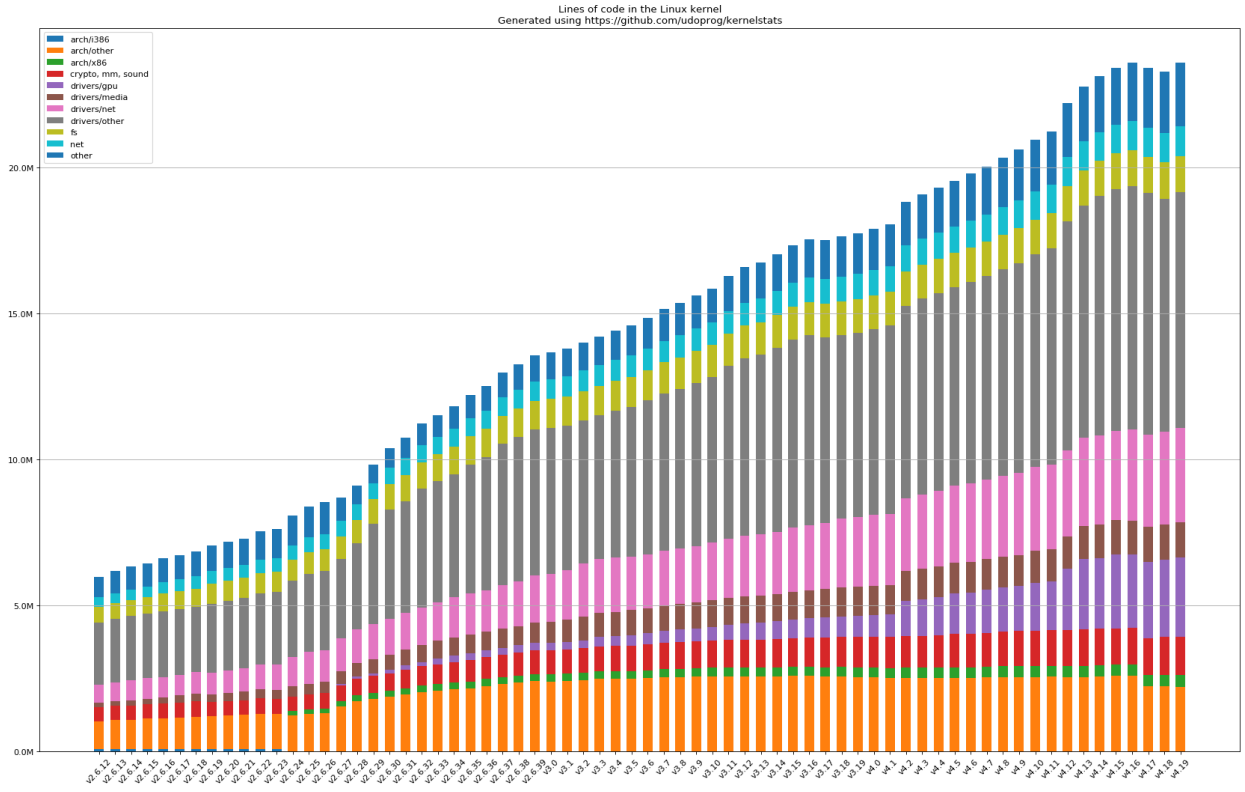


Figure 1.1: Lines of code in the Linux kernel.

However, prior work [1, 2, 3] shows that applications only need a small subset of kernel functionalities. For instance, often less than 100 system calls are used at a time by popular applications such as Nginx and Apache HTTP servers. More than 90% of the kernel, from the Ubuntu Linux distribution, size can be reduced, and a large part of the reduction comes

from `sound/`, `drivers/` and `net/`. As the web server does not play sounds, not require special hardware devices and only use a small subset of network protocols (*i.e.*, TCP/IP).

A large part of the kernel become unnecessary because applications only use a small subset of kernel functionalities. The unnecessary parts, bloat, in the kernel create serious security threats because the OS kernels are often considered to be a part of the trusted computing base (TCB) and the bloat creates new attack surfaces. Today, there exist many known exploits that take advantage of kernel vulnerabilities [4] to create significant losses of money or privacy. In addition to security issues, kernel bloat also imposes overheads and reduces system efficiency [5].

Research efforts that address OS kernel bloat include kernel rearchitecture and kernel debloating. Kernel rearchitecture builds a new kernel from scratch, and the new kernel is specialized for specific application workloads such as unikernels [6, 7, 8, 9, 10, 11]. Although kernel rearchitecture can achieve a highly specialized kernel, building a kernel from scratch often requires tremendous amount of engineering efforts, including porting existing applications and building missing features (network stacks or file systems), which hinder the practical usage of such approach. On the other hand, kernel debloating, sometimes referred as kernel specialization[1], aims to remove the redundancy from the existing kernel.

Kernel debloating [1, 2, 3, 12, 13, 14, 15] techniques can often been viewed as a two-step process: (1) identify the necessary parts of kernel and (2) enforce the kernel to use only the necessary parts. Due to the complexity and size of the kernel, kernel analysis on identifying necessary code is often done by dynamic analysis such as tracing [3, 12, 14, 15]. A number of approaches are used to restrict the access to unnecessary kernel components including removing them directly [3, 12, 13, 14], leveraging hypervisors to shadow the kernel memory [2, 15] or applying control flow integrity (CFI) [16]. As shown by recent studies, kernel debloating can effectively reduce the size of the kernel by 50%–85% [3], attack surface by 50%–85% [3], and security vulnerabilities by 34%–74% [17].

This dissertation firstly evaluates kernel debloating techniques with unikernels (kernel rearchitecture) and concludes that kernel debloating has its irreplaceable advantages such as the support from the huge Linux kernel community, the ability to run legacy applications and better application performance compared with unikernel systems. Motivated by the first study, this dissertation explores kernel debloating techniques and strives to find the best trade-off for a *practical* and *effective* kernel debloating framework. Finally, this dissertation presents an advanced kernel debloating framework, Cozart, that is automatic and produces stable kernels, and I use Cozart as a vehicle to conduct the existing limitations of kernel

[1]This dissertation uses these two words interchangeably.

debloating and share these insights and experiences to shed light on future research and development of kernel debloating.

1.1 RESEARCH GOAL AND CHALLENGES

The work in this dissertation starts off by making the following hypothesis:

There exists a practical kernel debloating technique that produces stable kernels, and such technique is automatic and requires minimal human intervention.

The objectives of this dissertation are to explore debloating techniques and strive to find the best tradeoffs among them to achieve the goal of a practical debloating framework. The key questions I would like to answer are:

1. Is debloating commodity OS kernel still worth doing despite the existence of kernel rearchitecture techniques (*e.g.*, unikernels)?
2. What are the right tradeoffs when debloating a commodity kernel?
3. What are the existing limitations that hinder the practical adoption of kernel debloating?

The challenges to answering these questions reside in the complexity of commodity kernels, for instance, the Linux kernel has more than 25 millions of lines of code. As discussed, kernel debloating contains two steps (1) identify necessary parts and (2) enforce the kernel to use only these necessary parts. For identifying necessary parts, ideally, program analysis techniques can extract *complete* necessary parts from the kernel. However, the complexity of the kernel makes it difficult to conduct program analysis techniques. For enforcing the specialization, multiple approaches are proposed including binary rewriting [18], kernel configuration options [3, 13, 14] and hypervisors [2]. Each approach has its pros and cons such as the degree of reduction, the stability of debloated kernel and the required manual efforts. Carefully weighing these tradeoffs is necessary to answer these research questions and build a practical and effective debloating framework.

1.2 SUMMARY OF SOLUTIONS

Based on the aforementioned challenges, this dissertation is divided into three parts that are introduced in Chapter 3, 4 and 5.

This dissertation firstly studies the comparison between kernel rearchitecture and kernel debloating techniques to understand the pros and cons of these two approaches. Specifically, this dissertation proposes Lupine. Lupine is a Linux system that is specialized for applications via configuration to mimic unikernels and attempt to achieve unikernels benefits including small image, fast boot time, security, low memory footprint and great application performance. Lupine is evaluated against a variety of unikernel systems such as rumprun [6] and OSv [19]. The result suggests that Lupine, a specialized (debloated) Linux system, can perform as good as unikernels or even better and that Linux systems are irreplaceable because of its community and abundant applications.

From the first study, I learn that Linux is irreplaceable and has the potential to be debloated to be lightweight, secure and performant. I was motivated to continue to explore commodity kernel (Linux) debloating. This dissertation then presents MultiK and DKut. MultiK is a in-kernel framework to orchestrate multiple specialized kernels for different applications. DKut is a kernel profiling technique that can collect necessary kernel parts for an application at the basic-block granularity. Combing MultiK and DKut achieves highly debloated kernels *i.e.*, at the basic-block granularity. My experiment result shows that an application only uses a small subset of kernel code *e.g.*, only 7% is used for a Web server. I also observe kernel instability during experiments when aggressive debloating is conducted.

Finally I learn that aggressive and intrusive kernel debloating leads instability and cause kernel crashes, therefore, hindering the practical adoption of kernel debloating. To build a kernel debloating framework that produces stable kernels, I use kernel configuration as the mechanism for debloating and build Cozart. Cozart is a practical framework because it is effective (more than 80% kernel reduction), automatic (no human intervention required) and generates stable kernels (the debloated kernels are validated to work as expected). I use Cozart as a vehicle to study how to make debloating more practical. My studies lead to the following findings and results:

- Existing kernel debloating techniques initialize in-kernel tracing methods (*e.g.*, ftrace) too late and cannot observe the boot phase, which is critical to producing a bootable kernel.
- Kernel debloating can be done within tens of seconds if the configuration options of target applications are known.
- Using instruction-level tracing (instead of `ftrace`) can address kernel configuration options that control intra-function features.

- An essential limitation of using dynamic-tracing based techniques for kernel debloating (which are the *de facto* approach) is the imperfect test suites and benchmarks.
- Domain-specific information can be used to further debloat the kernel by removing the kernel modules that were executed in the baseline kernel but are not needed by the actual deployment.
- Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and extensively customized kernels (e.g., the Firecracker kernel).

In addition to validation of my research hypervisor, I expand the traditional definition of kernel debloating. Besides removing redundant code from kernels, restricting access of redundant code and reducing inefficiency can also be a mean of debloating. I present KFuse, which reduces the inefficiency of existing kernel components (BPF).

I make the observation that extensions can be collectively optimized after they are verified individually and loaded into the shared kernel.

CHAPTER 2: BACKGROUND AND RELATED WORK

I discuss and compare different kernel debloating approaches. They can be broadly classified into (1) feature-based debloating, (2) compiler-based debloating, (3) binary debloating or (4) kernel rearchitecture.

Feature-based Debloating The Linux kernel provides the `KConfig` mechanism for configuring the kernel. However, the complexity of `KConfig` makes it hard to tailor a kernel configuration for a given application. Kurmus *et al.* [3] tried to automate `KConfig` based kernel customization by obtaining a runtime kernel trace for a target application, then mapping the trace back to the source lines and using source lines along with configuration dependencies for arriving at an optimal configuration. While they were able to achieve 50-80% reduction in kernel size, their approach still requires some manual effort for creating predefined blacklists and whitelists of configurations [20]. Further, when multiple applications need to be run, their approach creates a customized kernel for all of the applications together thereby limiting the effectiveness of the attack surface reduction achieved [18]. More recently, LightVM [21] tries to address bloat in the kernel by implementing a tool called TinyX that starts from Linux's `tinyconfig` and iteratively adds options from a bigger set of configuration options. This involves maintaining a manually produced white-list or black-list.

Compiler-based Debloating Modern compilers are much better at code optimization than humans are. A series of LWN.net articles [22, 23, 24, 25] discusses various cutting edge efforts in compiler and link-time techniques that are being developed in the Linux community that can eliminate significant amount of dead code and perform various other code optimizations. Most of the work in this area is experimental and does not produce a working kernel yet. They exist as out-of-tree patches [26]. The main challenge in applying these techniques to the Linux kernel arises out of the complexities in the kernel itself. Hand-written assembly, non-contiguous layout of functions, *etc.* do not make the kernel a good candidate for compiler-based optimization/specialization as is. It requires manually going through pieces of code and making careful changes without causing unexpected side effects. The LLVM community in recent years has produced a suite of advanced compiler tools [27]. The Linux community hasn't been able to take advantage of these due to its tight coupling with the gcc toolchain, making it hard to use other compilers like clang [28] from the LLVM tool chain to build the kernel. These are being fixed one patch at a time [29].

Binary Debloating Binary debloating techniques do not require reconfiguring or rebuilding the kernel. They work on the final kernel binary as is. KASR [2] specializes the kernel binary using a VM-based approach wherein they trace all the pages in the memory that are used by an application – for a few iterations until the trace doesn't change. This data is used to mark the unused pages in the extended page tables as non-executable, thus making the memory region unavailable to the application. Face-Change [18], shadow kernels [30] create specialized kernel text areas for each of the target applications and switch between them using a hypervisor to support multiple applications running together. Their performance is limited by the performance of the hypervisor. Face-Change reported performance overheads $\approx 40\%$ for I/O benchmarks and doesn't support multithreading. KASR customizes the kernel at a page level granularity.

Kernel Rearchitecture An orthogonal direction to specializing general purposes kernels for attack surface reduction is to use unikernels or microkernels that define a completely new architecture. Unikernels [10] get rid of protection rings and have the application code and the kernel in a single ring to reduce performance overhead arising from context switches. But this leaves kernel code that is required for the entire system including boot and termination available to the application. Microkernels such as Mach [31] design the kernel in a very modular manner such that the kernel TCB is minimal. This comes at the cost of performance overhead from context switching. Moreover both these approaches require the application to be re-built for the respective architectures. NOOKS [32] redesigns the kernel to isolate device drives from the kernel core – to protect from vulnerable device drivers. It still leaves active a major part of the kernel (for boot, shutdown and other OS tasks).

CHAPTER 3: DEBLOATED KERNELS AND UNIKERNELS

3.1 OVERVIEW

A unikernel is a specialized system stack constructed by library operating systems [10]. Unikernels are known for their promised performance characteristics such as small image size, fast boot time, low memory footprint and application performance.

In this chapter, I intend to study the unikernels and debloated commodity OS kernels (*e.g.*, Linux) to understand the advantage of debloating a commodity kernel against rebuilding a specialized kernel from scratch. I present Lupine, in which I apply two-well known unikernel-like techniques to Linux: configuration specialization and the elimination of system call overhead. Lupine can achieve all benefits that unikernels enjoys including small image size, fast boot time, low memory footprint and application performance.

I conduct evaluation for Lupine and various unikernel systems and conclude that debloated commodity OS kernels can perform as good as unikernels. More importantly, Lupine exploits Linux to eliminate the application compatibility issues of other unikernels; it can run any application using Linux's highly-optimized implementation, including those that do not fit in the unikernel domain.

3.2 INTRODUCTION

Since the inception of cloud computing, the virtual machine (VM) abstraction has dominated infrastructure-as-a-service systems. However, it has recently been challenged, as both users and cloud providers seek more lightweight offerings. For example, alternatives such as OS-level containers have begun to attract attention due (in part) to their relatively lightweight characteristics in dimensions such as image size, boot time, memory footprint and overall performance.

In response, the virtualization stack has been evolving to become more lightweight in two main ways. First, modern virtual machine monitor designs, like lightVM [21] and AWS Firecracker [33] (or in a more extreme case unikernel monitors [8]), have reduced the complexity and improved the performance of the monitor. Second, alternatives to large, general-purpose guest operating systems have begun to emerge—whether it is a change in the userspace (e.g., from Ubuntu-based to Alpine-based), a change in the configuration of the guest kernel (e.g., TinyX [21]) or in the case of *unikernels* [10], a specialized library OS.

Specializing the guest VM to the extent of running a library OS tailored to an application

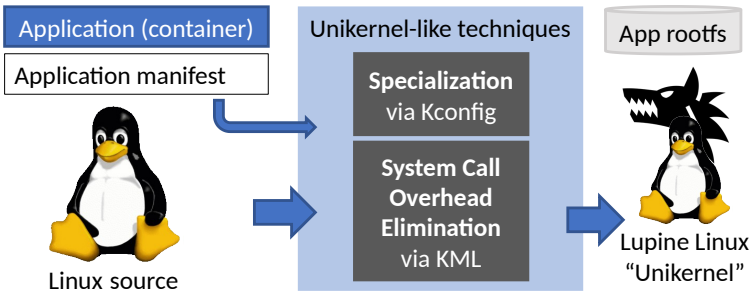


Figure 3.1: Overview

is a compelling prospect, especially when the application domain is limited. *Language-based* unikernels, such as MirageOS [10] (in which the library OS and application are entirely written in OCaml), have demonstrated a combination of security and lightweight properties. In an effort to expand the applicability of unikernel ideas, however, several unikernel/library OS projects, including HermiTux [34], Rumprun [6], Graphene [35] and OSv [19], have attempted to become more general and *POSIX-like*; some even going so far as claiming Linux binary compatibility [34]. Approaching some level of POSIX-like generality typically requires either a great implementation effort or a clever way to reuse existing POSIX-compatible kernel implementations (usually NetBSD [6, 19]). However, these approaches still fall short of true compatibility because of arbitrary restrictions (e.g., not supporting `fork`) and suffer by being unable to leverage the robustness, performance or, most importantly, the community of Linux.

In this paper, I make the observation that Linux is already highly configurable and seek to determine exactly how close it is to achieving the sought-after properties of unikernels. I describe *Lupine Linux* (Figure 3.1), in which I apply two well-known unikernel-like techniques to Linux: *specialization* and the *elimination of system call overhead*. Though I do not propose a general solution for specialization, I specialize Lupine through the kernel's Kconfig mechanisms by (1) eliminating functionality from the kernel that is not necessary for the unikernel domain (e.g., support for hardware devices or multiprocessing) and (2) tailoring the kernel as much as possible to the particular application. Lupine eliminates system call overhead by running the application in the same privilege domain as the kernel via the existing (but not upstream) Kernel Mode Linux (KML) [36] patch.

When evaluating Lupine against a state-of-the-art lightweight VM (AWS Firecracker's *microVM*) and three POSIX-like unikernels, I find that Lupine outperforms microVM and at least one of the reference unikernels in all of the following dimensions: image size (4 MB), boot time (23 ms), memory footprint (21 MB), system call latency (20 μs), and application performance (up to 33% higher throughput than microVM). While both unikernel techniques

played a role in improving performance, specialization had the largest effect: despite up to 40% reduction in system call latency due to KML on microbenchmarks, I found it improved application performance on macrobenchmarks by only 4%.

Regarding specialization via configuration, I attempted to determine the most practical degree of specialization for Lupine. To this end, I examined the effects of specialization in Lupine by *heuristically* creating specialized configurations for the top 20 cloud applications—that account for 83% of all downloads—as determined by popularity on Docker Hub. I categorize 550 configuration options from the microVM kernel and find only 19 of them are required to run all 20 applications, suggesting a tiny, application-agnostic kernel configuration that achieves unikernel-like performance without the complications of per-application specialization.

Lupine exploits Linux to eliminate the generality issues of other POSIX-like unikernels; it can run any application using Linux's highly-optimized implementation, including those that do not fit in the unikernel domain. In this context, I examine how unikernel properties degrade in the face of generality and find a *graceful degradation* property. Where other unikernels may crash on `fork`, Lupine continues to run. Moreover, I find virtually no overhead for supporting multiple address spaces and at worst an 8% overhead to support multiple processors, concluding that many unikernel restrictions are avoided unnecessarily for POSIX-like unikernels.

3.3 UNIKERNELS

Unikernels [6, 7, 10, 19, 34, 37, 38, 39, 40, 41] have garnered widespread interest by providing a lightweight, simple, secure and high-performance alternative to the complex, conventional, general-purpose compute stacks that have evolved over many years. In this section, I give a brief background and classification of unikernel projects and their benefits, describing some of the techniques that they have used to achieve these benefits and identify some common limitations.

3.3.1 Background

Unikernels are the most recent incarnation of library OS [35, 42, 43, 44] designs. They are typically associated with cloud environments and consist of a single application linked with a library that provides enough functionality to run directly on a virtual-hardware-like interface. I organize unikernels into two categories: *language-based* and *POSIX-like*.

Language-based. Language-based unikernels are library OS environments that are tied to a specific programming language runtime and libraries, for example, MirageOS [10] for OCaml, IncludeOS [37] for C++, Clive [38] for Go, HalVM [39] for Haskell, runtime.js [7] for JavaScript, Ling [40] for Erlang, and ClickOS [41] for Click router rules [45]. Language-based unikernels typically do not need to implement or adhere to POSIX functionality, which can lead to small images, suitability for compiler-based analyses and optimizations, and reliability or security from the use of language features. For example, 39 out of 40 of all bugs found in Linux drivers in 2017 were due to memory safety issues [46] that could have been avoided by a high-level language. However, the requirement for applications to adhere to a particular language and interface limits adoption.

POSIX-like. POSIX-like unikernels are library OS environments that use a single address space and a single privilege level but attempt to provide some amount of compatibility with existing applications. OSv [19] and HermiTux [34] are two unikernels that boast binary compatibility with Linux applications but reimplement kernel functionality from scratch, losing the opportunity to benefit from the maturity, stability, performance and community of Linux. Unlike language-based unikernels and Rumprun [6], a POSIX-like unikernel that leverages NetBSD to avoid reimplementation, OSv and HermiTux do not require the application to be linked with the library OS which (mostly) eliminates the need to modify application builds[1] and eases deployment at the cost of losing specialization opportunities.

3.3.2 Benefits and Techniques

Unikernels achieve benefits like low boot times, security, isolation, small image sizes, low memory footprint and performance through a combination of optimizing the monitor [8, 21] and construction of the unikernel itself.[2]

Lightweight monitors. Traditional virtual machine monitors like QEMU are general and complex, with 1.8 million lines of C code and the ability to emulate devices and even different CPU architectures. Recently, unikernel monitors [8] have shown that a unikernel's reduced requirement for faithful hardware emulation can result in a dramatically simpler, more isolated and higher performing monitor (which may not even require virtualization hardware [9]). As a result, unikernels have been shown to boot in as little as 5-10 ms, as

[1] These systems typically maintain a curated application list. While modifications to the applications on the list are relatively minor, this approach severely limits what can run in practice, as I will see in Section 3.5.

[2] Some unikernels, especially language-based unikernels, use other techniques, discussed further in Section 3.7.

opposed to hundreds of milliseconds for containers or minutes for VMs [8, 47], which is important for new compute models like serverless computing [48, 49]. At the same time, general-purpose monitors have also been reducing generality (such as forgoing some device emulation) for performance: for example, AWS Firecracker [33] and LightVM [21] optimize for boot time by eliminating PCI enumeration. Firecracker also improves the security posture of monitors by using a memory-safe language (Rust).

Specialization. Unikernels embody a minimalist philosophy, where they only require those code modules that are needed for a particular application to run on virtual hardware. This leads to smaller image sizes, lower memory footprint, and smaller attack surfaces. In language-based unikernels, like MirageOS, the relatively tight integration between the language, package manager and build process implements this philosophy well. POSIX-like unikernels, like Rumprun, OSv or HermiTux, balance how much they specialize with compatibility and reuse of legacy code, but typically attempt to provide at least coarse-grained specialization.

System Call Overhead Elimination. Unikernels contain, by definition, a single application. Therefore, logically, the library OS and the application exist in the same security and availability domain. As a result, unikernels typically run all code in the same CPU privilege domain and in the same address space in order to improve performance over traditional systems. There is no need to switch context between application and kernel functions.

3.3.3 The Ideal Unikernel

Unfortunately, existing unikernels face challenges in terms of generality; applications may need to be written from scratch, potentially in an unfamiliar language (e.g., language-based unikernels). Those that do try to address generality (e.g., POSIX-like unikernels) find themselves in the unenviable position of trying to reimplement Linux, at least in part.

The ideal unikernel would enjoy all of the benefits that they are known for while also being able to support Linux applications and share its community.

3.4 LUPINE LINUX

I make Linux behave like a unikernel by *specialization* and *system call overhead elimination*. I specialize Lupine according to both applications and the unikernel constraints such as the single-process nature or expected deployment environments (Section 3.4.1). I eliminate

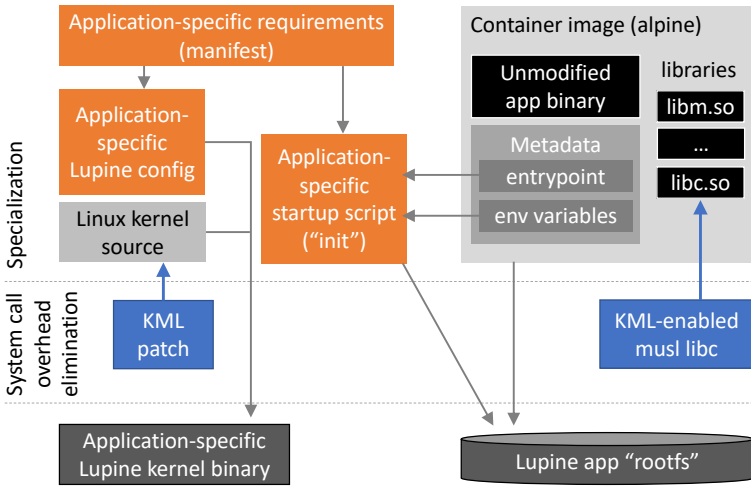


Figure 3.2: Specialization and system call overhead elimination in Lupine.

the system call overheads by applying KML patches to Linux that allow applications to run in kernel mode (Section 3.4.2).

Like some POSIX-like unikernels (e.g., HermiTux [34] and OSv [19]), but unlike others (e.g., Rumprun [6]), Lupine is not a single, statically-linked binary. Instead, a Lupine unikernel consists of a kernel binary that dynamically loads the application code from a root filesystem (rootfs) as in other Linux-based systems. Figure 3.2 shows the specifics of the generation of Lupine kernel binary and root filesystem. Lupine kernel binary is configured to be a small, special-purpose Linux kernel image, obtained via the specialization highlighted in orange in Figure 3.2. Also, the kernel is enhanced with Kernel Mode Linux (KML) so that Lupine runs the target application as a *kernel-mode process*, thereby avoiding context switches. An *application manifest* informs the application-specific kernel configuration.

I leverage Docker container images to obtain minimal root filesystems with applications and all their dependencies such as dynamically-linked libraries (e.g., `libc`, `libm`, etc.)[3]. As the root filesystem is specialized, Lupine does not employ a general-purpose `init` system. Instead, Lupine creates an application-specific startup script based on container metadata. For example, the *entrypoint* describes the parameters with which to invoke the application, and the *env variables* describe how to set up the environment. Like the kernel image, the script is informed by the application manifest; for example, it may initialize the network device, mount loopback devices or the proc filesystem, generate entropy, set ulimits, set environment variables, or create directories before executing the application. Finally, I convert the container images that include the application binary, a *KML-enabled* `libc` (described in Section 3.4.2) and the application-specific startup script into an `ext2` image that the spe-

[3]Tools such as Docker Slim [50] help ensure a minimal dependency set.

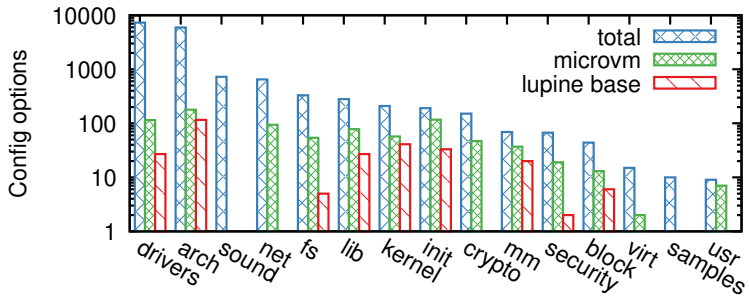


Figure 3.3: Linux kernel configuration options (log scale)

cialized Lupine kernel will use as its root filesystem. At runtime, a standard virtual machine monitor (e.g., Firecracker) launches the Lupine kernel and rootfs.

The concrete details of the application manifest are out of scope for this paper. At its simplest, an application manifest could be a developer-supplied kernel configuration and startup script.

3.4.1 Specialization

The Linux kernel contains considerable facilities for specialization through its `Kconfig` configuration mechanism. In total, there are 15,953 configuration options for Linux 4.0. Configuration options determine whether features should be included in the kernel by compiling the source code for the feature and either linking it into the kernel image or into a module that can be dynamically loaded into a running kernel. Kernel configuration options empower users to select or enable (for example) support for a variety of hardware (e.g., device drivers), a variety of services to applications (e.g., filesystems, network protocols) and algorithms governing management of the most basic compute resources (e.g., memory management, scheduling).

Figure 3.3 shows the total number of available configuration options (by directory) in the Linux source tree. Unsurprisingly, almost half of the configuration options are found in drivers to support the wide range of devices that Linux runs on. Figure 3.3 also shows the breakdown of configuration options selected by AWS Firecracker's microVM configuration. This is a Linux configuration that allows a general-purpose workload to specifically run on the Firecracker monitor on the `x86_64` architecture. This configuration can safely omit a vast majority of configurable functionality because of the known constraints of Firecracker, as shown in Figure 3.4. For example, the vast majority of the driver and architecture-specific options are not necessary since the virtual I/O devices and architecture are pre-determined.

Even more configurable functionality can be safely omitted for Lupine because of the

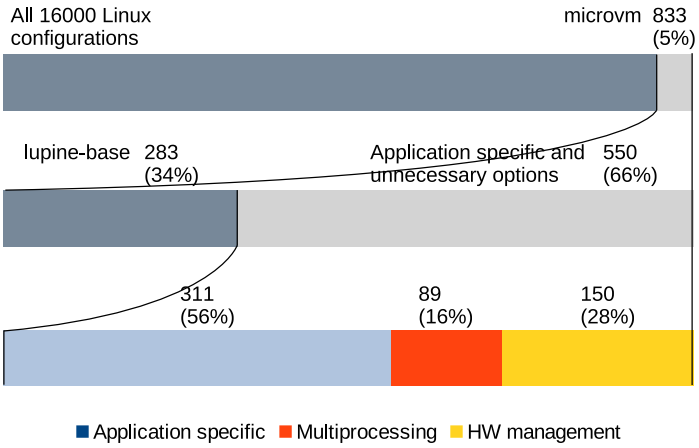


Figure 3.4: Breakdown of kernel configuration options down to unnecessary ones by unikernel property.

known constraints of the unikernel domain. As depicted in Figure 3.4, starting from Firecracker's microVM configuration, I manually removed approximately 550 (66%) of the selected options that I deemed potentially unnecessary for the unikernel domain as further described below. I refer to the remaining 283 (34%) configuration options as *lupine-base*.

I further manually classified the 550 removed options into categories based on features or design properties of unikernels and not based on the Linux kernel's structure as in Figure 3.3. *Application-specific* options are only necessary for certain applications and may be reintroduced on top of *lupine-base* to create an application-specific configuration. Others are not necessary for *any* unikernel application, either because of the single-process nature of unikernels or the predictable runtime environment of virtual machines in the cloud. I now describe these categories (Figure 3.4) and provide examples.

3.4.1.1 Application-specific options.

Unikernels are driven by a minimalist philosophy where they only contain functionality that the application needs. While compatibility with Linux often implies some compromises, an application-centric approach can be applied towards Linux kernel configuration. To this end, I categorize certain configuration options as application-specific, which may or may not appear in any Lupine unikernel's kernel configuration. I also discuss various granularities at which an application manifest could inform kernel configuration, but leave the generation of such a manifest (which could utilize static or dynamic analysis [3, 51, 52]) to future work.

Unikernels embody *DevOps* industry trends, in which system configuration and runtime operations tasks are tightly integrated with application development. I identified approx-

Option	Enabled System Call(s)
ADVISE_SYSCALLS	madvise, fadvise64
AIO	io_setup, io_destroy, io_submit, io_cancel, io_getevents
BPF_SYSCALL	bpf
EPOLL	epoll_ctl, epoll_create, epoll_wait, epoll_pwait
EVENTFD	eventfd, eventfd2
FANOTIFY	fanotify_init, fanotify_mark
FHANDLE	open_by_handle_at, name_to_handle_at
FILE_LOCKING	flock
FUTEX	futex, set_robust_list, get_robust_list
INOTIFY_USER	inotify_init, inotify_add_watch, inotify_rm_watch
SIGNALFD	signalfd, signalfd4
TIMERFD	timerfd_create, timerfd_gettime, timerfd_settime

Table 3.1: Linux configuration options that enable/disable system calls.

imately 100 network-related options, including a variety of less popular protocols and 35 filesystem-related configuration options that represent system configuration tradeoffs that depend on the (single) application. At a finer granularity, if I assume the application or container manifest details exactly which system calls an application will use,[4] then I can configure Linux to include some necessary system calls. For example, Table 3.1 lists configuration options that dictate whether one or more system calls (and their implementations) are compiled into the kernel binary. As an example of application-specific configuration, the `redis` key-value store requires EPOLL and FUTEX by default, whereas the `nginx` web server additionally requires AIO and EVENTFD. A Lupine kernel compiled for `redis` does not contain the AIO or EVENTFD-related system calls.

In addition to the above, some applications expect other services from the kernel, for instance, the `/proc` filesystem or `sysctl` functionality. Moreover, the Linux kernel maintains a substantial library that resides in the kernel because of its traditional position as a more privileged security domain. Unikernels do not maintain the traditional privilege separation but may make use of this functionality directly or indirectly by using a protocol or service that needs it (e.g., cryptographic routines for IPsec). I marked 20 compression-related and 55 crypto-related options from the microVM configuration as application-specific. Finally, Linux contains significant facilities for debugging; a Lupine unikernel can select up to 65 debugging and information-related kernel configuration options from microVM's configuration.

In total, I classified approximately 311 configuration options as application-specific as shown in Figure 3.4. In Section 3.5, I will evaluate the degree of application specialization via

[4]While generating the manifest is, in general, an open problem, several products and projects like DockerSlim[50] and Twistlock[53] rely on similar system-call information.

Linux kernel configuration (and its effects) achieved in Lupine for common cloud applications.

3.4.1.2 Unnecessary options.

Some options in microVM's configuration will, by definition, never be needed by any Lupine unikernel so they can be safely eliminated. I categorize these options into two groups: (1) those that stem from the single-process nature of unikernels and (2) those that stem from the expected virtual hardware environment in the cloud.

Unikernels are not intended for multiple processes. The Linux kernel is intended to run multiple *processes*, thus requiring configurable functionality for synchronization, scheduling and resource accounting. For example, *cgroups* and *namespaces* are specific mechanisms that limit, account for and isolate resource utilization between processes or groups of processes. I classified about 20 configuration options related to cgroups and namespaces in Firecracker's microVM configuration.

Furthermore, the kernel is usually run in a separate, more privileged security domain than the application. As such, the kernel contains enhanced access control systems such as SELinux and functionality to guard the crossing from the application domain to the kernel domain, such as `seccomp` filters, all of which are all unnecessary for unikernels. More importantly, security options with a severe impact on performance are also unnecessary for this reason. For example, KPTI (kernel page table isolation [54]) forbids the mapping of kernel pages into processes' page table to mitigate the Meltdown [55] vulnerability. This dramatically affects system call performance; when testing with KPTI on Linux 5.0 I measured a 10x slowdown in system call latency. In total, I eliminated 12 configuration options due to the single security domain.

Linux is well equipped to run on multiple-*processor* systems. As a result, the kernel contains various options to include and tune SMP and NUMA functionality. On the other hand, since most unikernels do not support fork, the standard approach to take advantage of multiple processors is to run multiple unikernels.

Finally, Linux contains facilities for dynamically loading functionality through modules. A single application facilitates the creation of a kernel that contains all functionality it needs at build time.

Overall, I attribute the removal of 89 configuration options to the single-process— *"uni"*— characteristics of unikernels as shown in Figure 3.4 (under "Multiple Processes"). In Section 3.6, I examine the relaxation of this property.

Unikernels are not intended for general hardware. default configurations for Linux are intended to result in a general-purpose system. Such a system is intimately involved in managing hardware with configurable functionality to perform tasks, including power management, hotplug and driving and interfacing with devices. Unikernels, which are typically intended to run as virtual machines in the cloud, can leave many physical hardware management tasks to the underlying host or hypervisor. Firecracker's microVM kernel configuration demonstrates the first step by eliminating many unnecessary drivers and architecture-specific configuration options (as shown in Figure 3.3). Lupine's configuration goes further by classifying 150 configuration options—including 24 options for power management that can be left to the underlying host—as unnecessary for Lupine unikernels as shown in Figure 3.4.

3.4.2 Eliminating System Call Overhead

Kernel Mode Linux [36] is an existing patch to Linux that enables normal user processes to run in kernel mode, and call kernel routines directly without any expensive privilege transitions or context switches during system calls. Yet they are processes that, unlike kernel modules, do not require any change to the programming model and can take advantage of all system services for normal processes such as paging or scheduling.

While KML was designed for multiple applications to run, some as kernel-mode processes (identified by the path to the executable rooted at `/trusted`) and some as normal user-mode processes, the goal for Lupine is to mimic a unikernel that, by definition, only contains a single—privileged—application. As a result, I modify KML for Lupine so that all processes (of which there should be one) will execute in kernel mode. Note that, despite running the application with an elevated privilege level via KML, no kernel bypass occurs. Kernel execution paths enumerated due to system calls by an application remain identical regardless of whether KML is in use or not.

For the implementation of KML in Lupine, I applied the modified KML patch to the Linux kernel. I also patched *musl* `libc`, the `libc` implementation used by Alpine, for the distribution of Linux that I chose for the container images that form the basis of Lupine unikernel images. The patch is minimal: it replaces the `syscall` instruction used to issue a system call at each call site with a normal, same-privilege `call` instruction. The address of the called function is exported by the patched KML kernel using the `vsyscall`[5]. For

[5]The original KML design took advantage of the 32-bit kernel and dynamic behavior in `glibc` to entirely avoid modifications to `glibc`. In 32-bit mode, some versions of `glibc` would dynamically select whether to do the newer, faster `sysenter` x86 instruction to enter the kernel on a system call or use the older, slower `int 0x80` mechanism. The decision was made based on information exported by the kernel via the `vsyscall` mechanism (a kernel page exported to user space). KML introduced a third option, `call`

Name	monitor	kernel ver	kernel conf	userspace
MicroVM	Firecracker	4.0	microVM	Alpine 3.10
Lupine	Firecracker	4.0	lupine-base	Alpine 3.10

Table 3.2: Systems used to evaluate the Lupine unikernel.

most binaries that are dynamically linked, the patched `libc` can simply be loaded without requiring the recompilation of the binary. Statically linked binaries running on Lupine must be recompiled to link against the patched `libc`. Note that this is far less invasive than the changes required by many unikernels including not only recompilation but often a modified build.

3.5 EVALUATION

The purpose of this evaluation is to show that Lupine can achieve most benefits of unikernels: small image size, fast boot time, small memory footprint, no system call overheads, and application performance. I compare several unikernel and non-unikernel systems as summarized in Table 3.2. *MicroVM* is a baseline, representing a state-of-the-art VM-based approach to running a Linux application on the cloud. *OSv*, *HermiTux* and *Rump* are unikernels that (partially) recreate Linux functionality inside their library OS. They provide comparison targets to define unikernel-like functionality for the purposes of the evaluation. All systems use the Firecracker monitor, except for HermiTux and Rump that use specialized unikernel monitors [8].[6]

I use the same Linux kernel version for all cases. I use Linux 4.0 with KML patches applied.[7] *MicroVM* uses AWS Firecracker's microVM configuration adapted to Linux 4.0. Lupine uses an application-specific configuration atop the microVM-derived *lupine-base* configuration, as described in Section 3.4.1, with 2 variants:

- *-nokml* is used to highlight the contribution of eliminating context switches versus specialization. *Lupine-nokml* differs from *lupine* in two ways: (1) it uses a kernel that does not have the KML patch applied, and (2) contains `CONFIG_PARAVIRT`, as also present in microVM, which unfortunately conflicts with KML despite being important for performance (as I will see).

- *-tiny* indicates Lupine is optimized for size over performance. *Lupine-tiny* differs from *lupine* in that it (1) is compiled to optimize for space with `-Os` rather than for perfor-

[6]Both `uhyve` and `solo5-hvt` are descendants of `ukvm`.
[7]Linux 4.0 is the most recent available version for KML.

mance with `-O2` and (2) has 9 modified configuration options that state clear space/performance tradeoffs in Kconfig (e.g., `CONFIG_BASE_FULL`).[8]

A third variant is not application-specific:

- *-general* is a Lupine kernel with a configuration derived from the union of all application-specific configurations from the most popular 20 applications as described in Table 3.3 in Section 3.5.1.

All experiments were run on a single server with an 8 core Intel Xeon CPU (E3-1270 v6) at 3.80GHz and 16 GB of memory. For a fair comparison, the unikernel (or guest) was limited to 1 VCPU (pinned to a physical core) as most unikernels are single-threaded and 512 MB of memory (except the experiment for memory footprint). This was done for all performance tests. The VM monitor could also make use of 3 additional CPUs and the benchmark client used the remaining 4 physical CPUs if a client was needed.

I present three main conclusions from the evaluation. First, I confirm that kernel specialization is important: Lupine achieves up to 73% smaller image size, 59% faster boot time, 28% lower memory footprint and 33% higher throughput than the state-of-the-art VM. However, I find that specialization on an application-level granularity may not be important: only 19 application-specific options cover the 20 most popular applications (83% of all downloads) and I find at most 4% reduction in performance by using a common configuration. Second, I find that, while running the application in the same privilege domain improves performance up to 40% on microbenchmarks, it only has a 4% improvement in macrobenchmarks, indicating that system call overhead should not be a primary concern for unikernel developers. Finally, I show that Lupine avoids major pitfalls of POSIX-like unikernels that stem from not being Linux-based, including both the lack of support for unmodified applications and performance from highly-optimized code.

3.5.1 Configuration Diversity

Lupine attempts to mimic the only-what-you-need approach of unikernels in order to achieve some of their performance and security characteristics. In this subsection, I evaluate how much specialization of the Linux kernel occurs in practice when considering the most popular cloud applications. My primary finding is that many of the same configuration options are required by the most popular applications, and they are relatively few (19 options for the 20 most popular applications).

[8] Determining exactly which options should be selected for a tiny kernel is difficult, but studies have shown that `tinyconfig` is a good starting point [56].

Name	**Downloads**	**Description**	**# Options atop *lupine-base***
nginx	1.7	Web server	13
postgres	1.6	Database	10
httpd	1.4	Web server	13
node	1.2	Language runtime	5
redis	1.2	Key-value store	10
mongo	1.2	NOSQL database	11
mysql	1.2	Database	9
traefik	1.1	Edge router	8
memcached	0.9	Key-value store	10
hello-world	0.9	C program "hello"	0
mariadb	0.8	Database	13
golang	0.6	Language runtime	0
python	0.5	Language runtime	0
openjdk	0.5	Language runtime	0
rabbitmq	0.5	Message broker	12
php	0.4	Language runtime	0
wordpress	0.4	PHP/mysql blog tool	9
haproxy	0.4	Load balancer	8
influxdb	0.3	Time series database	11
elasticsearch	0.3	Search engine	12

Table 3.3: Top twenty most popular applications on Docker Hub (by billions of downloads) and the number of additional configuration options each requires beyond the *lupine-base* kernel configuration. [9]

Unlike other unikernel approaches, Lupine poses no restrictions on applications and requires no application modifications, alternate build processes, or curated package lists. As a result, I were able to directly run the most popular cloud applications on Lupine unikernels. To determine popularity, I used the 20 most downloaded container images from Docker Hub [57]. I find that popularity follows a power-law distribution: 20 applications account for 83% of all downloads. Table 3.3 lists the applications.

For each application, in place of an application manifest, I carried out the following process to determine the minimal viable configuration. First I ran the application as a standard container to determine *success criteria* for the application. While success criteria could include sophisticated test suites or achieving performance targets, I limited ourselves to

[9] I exclude the Docker daemon in this table because Linux 4.0 does not support layered file systems, a prerequisite for Docker.

the following tests. Language runtimes like `golang`, `openjdk` or `python` were tested by compiling (when applicable) a hello world application and testing that the message was correctly printed. Servers like `elasticsearch` or `nginx` were tested with simple queries or health status queries. `haproxy` and `traefik` were tested by checking the logs indicating that they were ready to accept traffic. I discuss the potential pitfalls of this approach in Section 3.7.

Once I had determined success criteria, I attempted to run the application on a Linux kernel built with the *lupine-base* configuration as described in Section 3.4.1. Recall that the base configuration is derived from microVM but lacks about 550 configuration options that I classified as hardware management, multiprocessing and application-specific. Some applications require no further configuration options to be enabled beyond *lupine-base*. For others, I added new options one by one while testing the application at each step. I expected all new options to be from the set classified as application-specific.

The process was manual: application output guided which configuration options to try. For example, an error message like *"the futex facility returned an unexpected error code"* indicated that I should add `CONFIG_FUTEX`, *"epoll_create1 failed: function not implemented"* suggested I try `CONFIG_EPOLL` and *"can't create UNIX socket"* indicated `CONFIG_UNIX`. Some error messages were less helpful and required some trial and error. Finally, some messages indicated that the application was likely not well-suited to be a unikernel. For example, `postgres` in Linux is made up of five processes (background writers, checkpointer, and replicator). It required `CONFIG_SYSVIPC`, an option I had classified as multi-process related and therefore not appropriate for a unikernel. Lupine can run such an application despite its obvious non-unikernel character, which is an advantage over other unikernel-based approaches. I will discuss the implications of relaxing unikernel restrictions in Section 3.6.

I conservatively estimate the time spent per application for a knowledgeable researcher or graduate student as 1 to 3 hours. However, I found knowledge of kernel options and experience accelerated the process. For example, I no longer need to perform trial and error for certain options, as I have learned that `CONFIG_FUTEX` is needed by glibc-based applications, and `CONFIG_EPOLL` is used by applications that utilize event polling.

Table 3.3 shows the number of configuration options (beyond *lupine-base*) deemed necessary to reach the success criteria for each application. Figure 3.5 depicts overlapping options thus showing how the union of the necessary configuration options grows as more applications are considered. The union of all configuration options is 19; in other words, a kernel (*lupine-general*) with only 19 configuration options added on top of the *lupine-base* configuration is sufficient to run all 20 of the most popular applications. The flattening of the growth curve provides evidence that a relatively small set of configuration options may

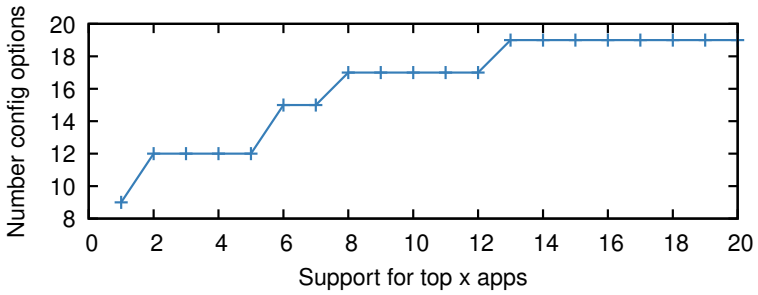


Figure 3.5: Growth of unique kernel configuration options to support more applications.

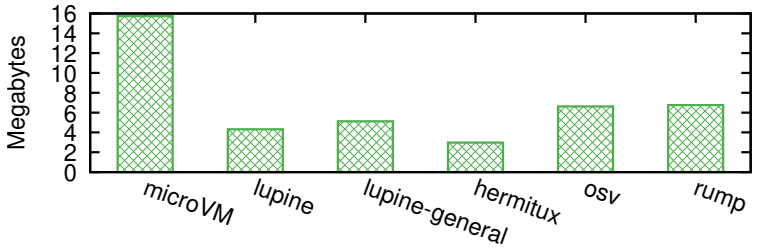


Figure 3.6: Image size for hello world.

be sufficient to support a large number of popular applications.

As I show later in the evaluation, a kernel containing all of these common options, *lupine-general*, performs similarly to a manually configured kernel, an observation that matches recent results from building a systematic kernel debloating framework [13]. As a result, general users will likely not need to perform the manual process described in this section and can use *lupine-general* directly. It is an open question, however, to provide a guarantee that *lupine-general* is sufficient for a given workload.

3.5.2 Image Size

Most unikernels achieve small image sizes by eschewing generality. Similarly, Lupine uses the Linux kernel's configuration facilities for specialization. Figure 3.6 compares the kernel image size of Lupine to microVM and several unikernels—all configured to run a simple hello world application in order to measure the minimal possible kernel image size. The *lupine-base* image (sufficient for the hello world) is only 27% of the microVM image, which is already a slim kernel image for general cloud applications. When configuring Lupine to optimize for size over performance (*-tiny*), the Lupine image shrinks by a further 6%.

Figure 3.6 shows Lupine to be comparable to my reference unikernel images. All configurations except Rump utilize dynamic loading, so I report only the size of the kernel. To avoid unfairly penalizing unikernels like Rump, that statically link large libraries (like `libc`,

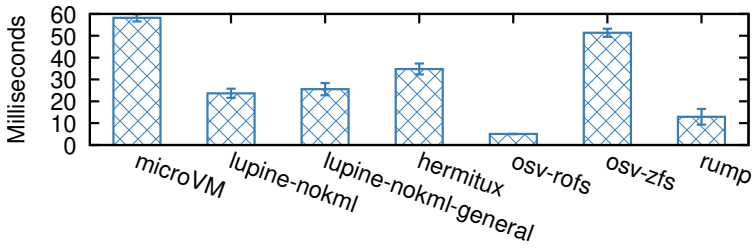


Figure 3.7: Boot time for hello world.

which consists of 24M), I configure them to run a trivial hello world application without `libc`.

I also examined the effect on application-specific configuration on the Lupine kernel image size. I found that the image size of lupine kernels varied from 27−33% of microVM's baseline. Compared to *lupine-base*, this corresponds to an increase of up to 19 percent. Even with the largest Lupine kernel configuration (*lupine-general*, that is capable of running all of the top 20 applications) the resulting image size remains smaller than the corresponding OSv and Rump image sizes. I note that *lupine-general* is an upper bound for kernel image size for the kernels associated with any application in Table 3.3, including `redis`, `nginx`, etc.

3.5.3 Boot Time

Figure 3.7 shows the boot time to run a hello-world application for each configuration. Firecracker logs the boot time of all Linux variants and OSv based on an I/O port write from the guest. I modified the unikernel monitors `solo5-hvt` and `uhyve` respectively to similarly measure boot time via an I/O port write from the guest.

As shown in Figure 3.7, use of a unikernel monitor does not guarantee fast boot time. Instead, unikernel implementation choices dominate the boot time. The OSv measurements show how dramatic the effects of unikernel implementation can be: when I first measured it using zfs (the standard r/w filesystem for OSv), boot time was 10x slower than the numbers I had seen reported elsewhere. After investigation, I found that a reduction in unikernel complexity to use a read-only filesystem resulted in the 10x improvement, thus underscoring the importance of implementation.

Lupine's configuration shows significant improvement over microVM and comparable boot time to the reference unikernels. In Figure 3.7, I present the boot time without KML (*lupine-nokml*). A primary enabler of fast boot time in Linux comes from the `CONFIG_PARAVIRT` configuration option which is active in microVM and *lupine-nokml*, but currently incompatible with KML. Without this option boot time jumps to 71 ms for Lupine. I believe that the

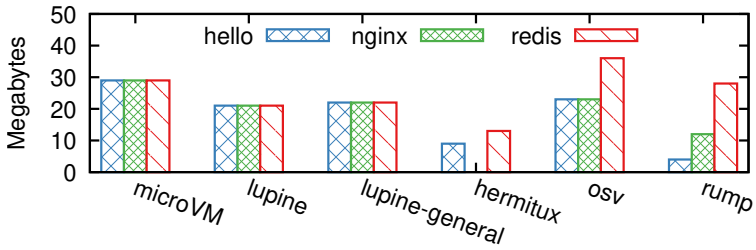


Figure 3.8: Memory footprint.

incompatibilities with KML are not fundamental and could be overcome with engineering effort and would result in similar boot times to *lupine-nokml*. I do not find an improvement in Lupine's boot time when employing space-saving techniques (*-tiny*) with or without KML. In other words, the 6% reduction in image size described in Section 3.5.2 does not affect boot time thus implying that boot time is more about reducing the complexity of the boot process than the image size. For *lupine-general*, I measured an additional boot time of 2 ms. Note that this is still faster than HermiTux and OSv (with `zfs`). I note that, similar to image size, *lupine-general* conveys an upper bound in kernel boot time for the kernels associated with any application in Table 3.3, including `redis`, `nginx`, etc.

3.5.4 Memory Footprint

Unikernels achieve low memory footprint by using small runtime images that include only what is needed to run a particular application. I define the memory footprint for an application as the minimum amount of memory required by the unikernel to successfully run that application as defined by success criteria described in Section 3.5.1. I determine the memory footprint by repeatedly testing the unikernel with a decreasing memory parameter passed to the monitor. My choice of applications was severely limited by what the (non-Lupine) unikernels could run without modification; I only present the memory footprint for three applications as shown in Figure 3.8. Unfortunately, HermiTux cannot run `nginx`, so I omit that bar.

Figure 3.8 shows the memory footprint for each application. In both application-specific and general cases, Lupine achieves a comparable memory footprint that is even smaller than unikernel approaches for `redis`. This is due in part to lazy allocation. While each of the unikernels shows variation in memory footprint, the Linux-based approaches (microVM and Lupine) do not.[10] There is no variation because the Linux kernel binary (the first binary to

[10]OSv is similar to Linux in this case in that it loads the application dynamically, which is why `nginx` and `hello` exhibit the same memory footprint; I believe `redis` exhibits a larger memory footprint because of how the OSv memory allocator works.

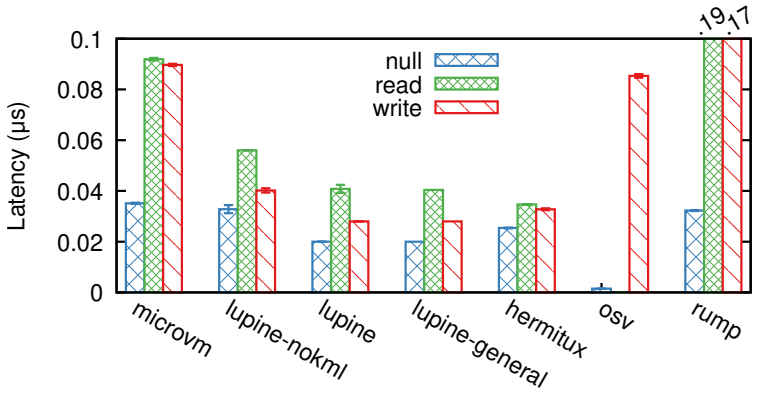


Figure 3.9: System call latency via `lmbench`.

be loaded) is more or less the same size across applications. The binary size of the application is irrelevant if much of it is loaded lazily and even a large application-level allocation like the one made by `redis` may not be populated until later. However, an argument can be made in favor of eliminating laziness and upfront knowledge of whether sufficient resources will be available for an application. I further discuss this issue in the context of immutable infrastructure in Section 3.7.

3.5.5 System call latency microbenchmark

Unikernels claim low system call latency due to the fact that the application is directly linked with the library OS. Using Lupine, a Linux system, can achieve similar system call latency as other POSIX-like unikernel approaches. Figure 3.9 shows the `lmbench` system call latency benchmark for the various systems. [11] The results show that Lupine is competitive with unikernel approaches for the null (`getppid`), read and write tests in `lmbench`. OSv shows the effects of implementation choices as `getppid` (issued by the *null* system call test) is hardcoded to always return 0 without any indirection. Read of `/dev/zero` is unsupported and write to `/dev/null` is almost as expensive as the microVM case.

Experimentation with *lupine-nokml* shows that both specialization and system call overhead elimination play a role. Specialization contributes up to 56% improvement (achieved during the write test) over microVM. However, I found no differences in system call latency between the application-specific and general variants (*lupine-general*) of Lupine. KML provides Lupine an additional 40% (achieved during the null test) improvement in system call latency over *lupine-nokml*.

[11] I only use the system call latency benchmark in `lmbench` due to lack of support in some unikernels for more complex benchmarks.

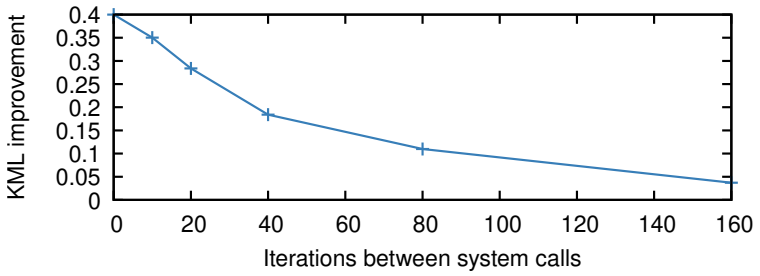


Figure 3.10: Relationship of KML syscall latency improvement to busying-waiting iterations (the more busy-waiting iterations the less frequent of user-kernel mode switching).

To better understand the potential performance improvements of KML on Lupine, I designed a microbenchmark in which I issued the null (`getppid`) system call latency test in a loop, while inserting a configurable amount of CPU work via another tight loop to control the frequency of the switching between user and kernel mode: the frequency of switching decreases as the number of iterations increases.

In an extreme case where the application calls the system call without doing anything else (0 iterations) KML provides a 40% performance improvement. However, Figure 3.10 shows how quickly the KML benefits are amortized away: with only 160 iterations between the issued system calls the original 40% improvement in latency drops below 5%. I find similarly low KML benefits for real-world applications in Section 3.5.6.

3.5.6 Application performance

Name	**redis-get**	**redis-set**	**nginx-conn**	**nginx-sess**
microVM	1.00	1.00	1.00	1.00
lupine-general	1.19	1.20	1.29	1.15
lupine	**1.21**	**1.22**	**1.33**	**1.14**
lupine-tiny	1.15	1.16	1.23	1.11
lupine-nokml	1.20	1.21	1.29	1.16
lupine-nokml-tiny	1.13	1.13	1.21	1.12
hermitux	.66	.67		
osv			.87	.53
rump	.99	.99	1.25	.53

Table 3.4: Application performance normalized to MicroVM (Note: higher value is better).

Unikernels boast good application performance due to lack of bloat and the elimination of system call latency. Table 3.4 shows the throughput of two popular Web applications: the `nginx` web server and the `redis` key-value store, normalized to microVM performance. As

in the memory footprint experiment in Section 3.5.4, I were severely limited in the choice of applications by what the various unikernels could run without modification.

For clients, I used `redis-benchmark` to benchmark two common `redis` commands, `get` and `set`, measuring requests per second. For `nginx`, I used `ab` to measure requests per second. Under the connection-based scenario (*nginx-conn*), one connection sends only one HTTP request. Under the session-based scenario (*nginx-sess*), one connection sends one hundred HTTP requests.[12] I ran the clients on the same physical machine to avoid uncontrolled network effects.

As shown in Table 3.4, Lupine outperforms the baseline and all the unikernels. A general kernel (*lupine-general*) that supports 20 applications in Section 3.3 does not sacrifice application performance. I note that, as a unikernel-like system with a single trust domain, Lupine does not require the use of many recent security enhancements that have been shown to incur significant slowdowns, oftentimes more than 100% [5]. I attribute much of Lupine's 20% (or greater) application performance improvement (when compared to baseline) to disabling these enhancements. The poor performance of the unikernels is most likely due to the fact that the implementation of kernel functionality in Linux has been highly optimized over many years thanks to the large Linux community, beyond what other implementations can achieve. I would like to have more data points, but the inability to run applications on the unikernels is a significant challenge: even with these two extremely popular applications, OSv drops connections for `redis` and `nginx` has not been curated for HermiTux.

Within the Lupine variants, optimizing for space (e.g., *-tiny*) can cost up to 10 percentage points (for *nginx-conn*), while KML adds at most 4 percentage points (also for *nginx-conn*). As in the other experiments, KML and optimizing for size affects performance only a small amount relative to specialization via configuration.

3.6 BEYOND UNIKERNELS

Unikernel applications (and their developers) are typically restricted from using multiple processes, processors, security rings and users. These restrictions are often promoted as a feature (e.g., a single address space saves TLB flushes and improves context-switch performance [19, 34]) and justified or downplayed in certain contexts (e.g., many microservices do not utilize multi-processing [9]). Unfortunately, there is no room for bending the rules: as a unikernel, an application that issues `fork` will often crash or enter into an unexpected state by a stubbed-out `fork` implementation (e.g., continuing as a child where there is no parent).

[12] I use the `--keepalive` option in `ab`.

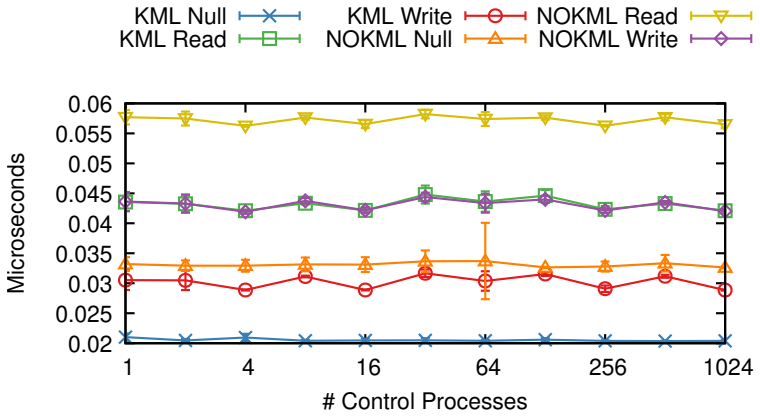


Figure 3.11: System call latency with different number of background control processes for KML and NOKML.

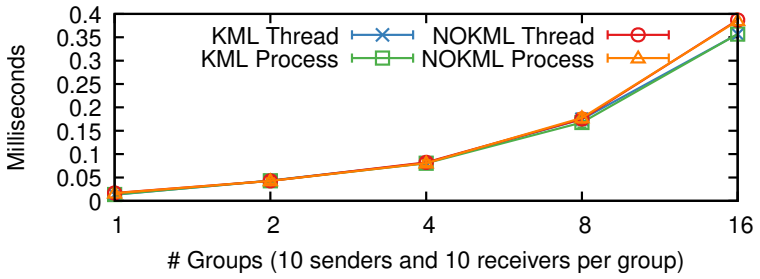


Figure 3.12: Perf context switch benchmark with threads and processes.

Such rigidity leads to serious issues for compatibility: as I encountered in my evaluation, it is unlikely that an existing application will run unmodified on a unikernel, even if the library OS is more-or-less binary compatible. Furthermore, there are situations where relaxing the unikernel restrictions is imperative. As a trivial example, building the Linux kernel with a single processor takes almost twice as long as with two processors.

Lupine is, at its core, a Linux system, and relaxing its unikernel properties is as simple as re-enabling the relevant configuration options. This results in a *graceful degradation* of unikernel-like performance properties. For example, rather than crashing on `fork`, Lupine can continue to execute correctly even if it begins to experience context switch overheads. Next, I investigate what the cost would be for Lupine to support applications that use multiprocessing features and whether including this support would adversely affect applications that do not.

I first consider the use of multiple address spaces and experiment with two different scenarios. First, I consider auxiliary processes that spend most of their time waiting either waking up or running in a frequency that does not interfere or create contention on resources with the application. I refer to such processes as *control processes*, i.e., processes that are respon-

sible for monitoring the application for multiple purposes (e.g., shells, environment setup, recovery and analysis, etc.). In practice, it is extremely common, for example, to find a script that forks an application from a shell after setting up some environment variables. Lack of support for this case from existing POSIX-compliant unikernel implementations severely limits their generality. I design an experiment to show that such uses of multiple address spaces are not harmful to unikernel-like performance. Specifically, I measure the system call latencies after launching 2^i $(i = 0, 1, \ldots, 10)$ control processes, using `sleep` as the control process. As shown in Figure 3.11, in all cases, there is no latency increase; all measurements (averaged over 30 runs) are within one standard deviation.

Second, I consider co-existing processes that contend resources with each other and may experience context switch overheads. To quantify these overheads, I compare the context switch overheads for threads that do not switch address spaces (to approximate unikernel behavior) versus processes. I use the messaging benchmark in `perf` [58] where 2^i $(i = 0, 1, 2, 3, 4)$ groups (10 senders and 10 receivers per group) of threads or processes message each other via UNIX sockets. The benchmark implements threads with `pthread` and processes with `fork`. For each configuration, I average the results of 30 runs. As shown in Figure 3.12, surprisingly, in all numbers of groups, switching processes is not slower than switching threads as the maximum time increase is 3% (in the KML case) when there is 1 group. In some cases, I even see process switching outperforming thread switching by $0 - 4\%$. This finding matches prior work [59]. Since no performance is lost, I conclude that the adherence to a single address space is unfounded from a performance perspective

Next, I investigate the effects of Linux's symmetric multiprocessing (SMP) support as configured by the `CONFIG_SMP` kernel configuration option. I devised three experiments, *sem_posix*, *futex* and *make -j*, to show the worst-case scenario for supporting SMP: a system with one processor running applications that frequently context switch. I expect to see an overhead from a kernel that supports SMP versus a more unikernel-like kernel that does not. *sem_posix* and *futex* spawn up to 512 workers that rapidly exercise futex and POSIX semaphore wait and post operations. Each worker starts 4 processes sharing a futex or semaphore. *make -j* builds the Linux kernel using up to 512 concurrent processes. I found *sem_posix* incurs up to 3%, *futex* incurs up to 8%, and *make* incurs up to 3% overhead over a kernel without SMP support. In most cases, where there would be fewer context switches, I would expect even less overhead, so the choice to use SMP—rejected by unikernels—will almost always outweigh the alternative.

3.7 DISCUSSION

The evaluation in Section 3.5 and the opportunity to gracefully degrade for non-unikernel workloads described in Section 3.6 make a compelling case for Lupine over other general-purpose unikernel approaches. Here I discuss the robustness of my analysis of Lupine and some benefits that unikernels achieve that Lupine does not.

3.7.1 Threats to validity

The conclusions drawn from my evaluation of Lupine rely on the correctness of my methodology, mainly in how the Linux kernel configuration—a notoriously messy part of Linux—is treated. The main risk is that Lupine has underestimated the necessary kernel configuration options for each application.

First, when determining the *lupine-base* configuration in Section 3.4, I may have misclassified certain kernel options as unnecessary rather than application specific. Moreover, the minimum set of configuration options that make up *lupine-base* may not be unique especially when considering options that provide similar functionality with different performance or space tradeoffs such as different compiler flags: `-O2` and `-Os`. Even if I were to find a different, more minimal *lupine-base*, the conclusions would hold.

Deriving an application-specific kernel configuration is more concerning. While not a focus of this paper—I assume its existence in the form of an application manifest—the evaluation of Lupine depends on an accurate application-specific kernel configuration. I determined configurations for the top 20 applications on Docker Hub based on a manual process based on simple success criteria and benchmarks that allowed us to quickly evaluate the configurations for many applications. When considering applications that do one thing and do it well (e.g., microservices), it may be more feasible to have a complete test suite to ensure that all configuration options are accounted for. In general, the problem is difficult: a large body of ongoing work attempts to derive kernel configuration from an application [3, 51, 52]. However, I believe the risk to be low: I noticed that many applications perform a series of checks when they start up, reducing the importance of complex success criteria. In my experience, in all cases, a weaker success criteria based on console output matched the configurations derived based on benchmark success.

Finally, I note that language-based unikernels, such as MirageOS [10], while unable to run existing POSIX applications, can use language-level analyses and package management techniques to determine application dependencies on OS functionality (e.g., networking), essentially removing the need for a manifest as needed by Lupine.

3.7.2 Unachieved unikernel benefits

Two of the unikernels I evaluated Lupine against (HermiTux [34] and Rump [60]) run on unikernel monitors [8] that are slim and optimized for unikernel workloads. Beyond boot times, unikernel monitors have been demonstrated to require such a small amount of host functionality that they can be implemented even as processes, thereby increasing performance while maintaining isolation [9, 61]. Linux does not currently run on a unikernel monitor, but it may possibly in the future given the fact that Linux can run in a variety of limited hardware environments (e.g., even with no MMU) or in environments that are not hardware like (e.g., User Mode Linux [62]).

The concept of *immutable infrastructure* has been associated with unikernels—especially language-based unikernels—in part because they push tasks that are traditionally done at deploy-time or later (like application configuration) to build time. As a result, the unikernel is specialized not just for an application but for a particular deployment of an application. However, general-purpose systems and applications often have dynamic or lazy properties—such as those seen when measuring the memory footprint in Section 3.5.4—that limit how immutable deployments can be. For example, interpreted code like JavaScript has become popular in the cloud but is dynamically loaded and interpreted. Dynamic behavior—which is pervasive in existing cloud applications and programming models—and immutability will continue to participate in a fundamental tussle as the cloud landscape evolves.

Another benefit of language-based unikernels that Lupine does not enjoy is the ability to perform language-based analyses or compiler optimizations that span both the application and kernel domain. For instance, MirageOS can employ whole-system optimization techniques on the OCaml code—from application to device drivers. POSIX-like unikernels tend to have less opportunity for this type of compiler optimization due to practical deployment and compatibility concerns when attempting to support legacy applications without completely changing their build processes. In the case of Linux, while link-time-optimization (LTO) exists for the kernel, it does not include the application in the analyses. Specializing the kernel via configuration, as shown for Lupine, may improve the results of LTO, but kernel routines or system calls cannot be inlined into the application without modifying both the kernel and application build processes. While some interesting new approaches are attempting this in the context of Linux [63], simultaneously maintaining full generality, or the ability to run any application (as Lupine can), remains a challenge.

3.8 CONCLUSION

While unikernels and library OS designs seem like a reasonable, lightweight alternative to more traditional virtual machines, the desire to increase the generality of unikernels along with an underestimation of the versatility of Linux has led us to stray too far from the potential benefits of unikernels. I show that Lupine, a pure Linux system, can outperform such unikernels in all categories including image size, boot time, memory footprint and application performance while maintaining vital properties, e.g., the community and engineering effort in the past three decades. Future research efforts should focus on making Linux specialization more effective and accessible.

CHAPTER 4: KERNEL SPECIALIZATION ENFORCEMENT WITH TRADEOFFS

4.1 OVERVIEW

In this chapter, I intend to explore specialization techniques by developing DKut and SKut– two methods to profile and eliminate unwanted kernel code at different granularities. In order to support multiple specialized copies of kernel to run in the same operating system, I present MultiK. MultiK is a Linux-based framework that orchestrates multiple kernels that are specialized for individual applications in a transparent manner.

I evaluate the framework against benchmarks (STREAM [64] and perf [58]) and applications (Apache httpd [65]). The results reveal that MultiK incurs virtually zero performance overheads. When the kernel the specialized aggressively (*e.g.*, basic-block granularity), the amount of kernel code reduction in size is as high as 93% for httpd when the baseline is the ubuntu vanilla kernel. With a such aggressive kernel reduction, the produced kernel is highly unstable and likely to crash because necessary code is not captured during the profiling phase such as error-handling code. When reducing the kernel more inclusively at system call granularity, the amount of reduction drops 82%. The produced kernel is more stable than the one debloated at basic-block granularity although it still crashes sometimes.

In the next section, I introduce my study methodology including the framework, MultiK, that orchestrates specialized kernels and two kernel profiling techniques (DKut and SKut).

General-purpose operating systems (OSes) have, over time, bloated in size. While this is necessitated by the need to support a diverse set of applications and usage scenarios, a significant amount of the kernel code is typically not required for *any given application*. For example, Linux supports more than 300 system calls and contains code for supporting different filesystems, network protocols, hardware drivers, *etc.* all of which may not be needed for every application or deployment. While a minimal off-the-shelf install of Ubuntu 16.04 (running kernel 4.4.1) produces a kernel binary with an $8MB$ text section, many of the applications that I profiled (refer to Section 4.6 for more details) only use about $800KB$ of it.

In addition to performance issues, unused kernel code (when mapped into an application's process memory) represents an attack surface – especially if vulnerabilities exist in the unused parts of the kernel code. Such vulnerabilities, while less common than those in applications, are still found with regular frequency.[1] Since OS kernels are often considered to be a part of

[1]Over 2500 vulnerabilities have been found in the Linux Kernel since 2010 (`https://nvd.nist.gov/`).

the trusted computing base (TCB) for many systems, this attack surface poses a significant risk. Today, there exist many known exploits that take advantage of kernel vulnerabilities (*e.g.*, CVE-2017-16995[2]).

Researchers have explored different techniques to reduce kernel code (*e.g.*, [2, 3, 10, 18, 66]). For example, (1) building application specific *unikernels* [10], (2) tailoring kernels through build configuration editing [3, 66], (3) providing specialized kernel views for each application [2, 18] among others. These approaches, either need application level changes [10], or need expert knowledge about (and manual intervention in) the selection of configurations – they also sacrifice the amount of kernel reduction achieved to support multiple applications [3, 66], or incur significant performance overheads [18] or can only specialize the kernels at a coarse page level granularity [2]. Note: "granularity", in this context, refers to sizes of code chunks that are considered for elimination; some techniques eliminate kernel code at the page level [2] while others may choose to do it at a basic block level [18]. I show that my framework can evaluate systems with different levels of code reduction granularity (with the obvious result that with a finer granularity of code reductions, a greater amount of kernel code can be eliminated (Section 4.6.1)).

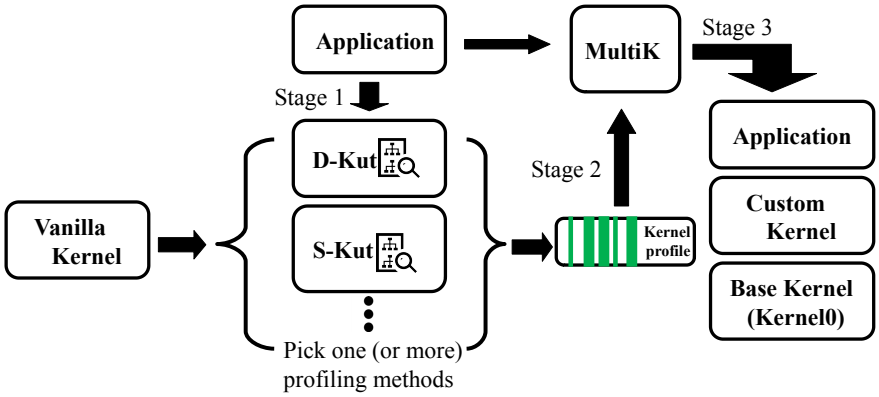


Figure 4.1: High level architecture of MultiK. **Stage 1**: an application is profiled to identify necessary kernel code using techniques chosen by developers (this paper presents two such possible techniques: DKut and SKut). **Stage 2**: MultiK reads the information for code reduction (based on the kernel profile) generated in Stage 1. **Stage 3**: MultiK creates the customized kernel (based on kernel profile from in Stage 2) when the application is launched.

In this paper I introduce MultiK (Section 4.2), *a flexible, automated, framework for (1) orchestrating multiple kernels (full or specialized) at (2) user chosen code reduction granularity and (3) near-zero runtime overheads* – without the need for virtualization. In addition, MultiK does not introduce additional overhead if a finer-grain granularity is chosen. To

[2] `https://access.redhat.com/security/cve/cve-2017-16995`

demonstrate this, I evaluate MultiK on three levels of granularity using two specialization tools (Section 4.4), (1) DKut, a run-time profiling-based tool that can tailor the Linux kernel at different granularity (I perform basic block- and function/symbol-granularity tailoring in this paper) and (2) SKut, a syscall-based tool that tailors the kernel based on the system calls used by the application. MultiK is able to simultaneously run multiple applications that either have their kernels specialized with any one of these tools or use the full, unmodified, kernel. Note that, though the two approaches (DKut and SKut) are complementary – they can either be used independently or in conjunction with other tools/frameworks[3].

Figure 4.1 shows the high-level architecture of my MultiK framework. A vanilla kernel is profiled (Stage 1) using the application that is supposed to run on it. Note that in contrast to existing approaches, the system designer can choose the level of granularity and code reduction techniques (Stage 2). At runtime (Stage 3), the MultiK framework launches the application on its specialized kernel (one application per specialized kernel[4]). The entire framework executes on a "base" (vanilla) kernel. The details of each of these stages/components are explained in the rest of the paper.

MultiK imposes very small, almost negligible, runtime overheads ($< 1\%$) *regardless of the granularity chosen* (Section 4.6). User programs run unmodified and natively. In fact, the entire framework is transparent to application developers. My framework can also easily integrate with container-based systems such as Docker [5] – which are popular for deploying applications.

In summary, this paper makes the following contributions:

1. Design and implement MultiK (Section 4.3), a flexible *framework* for dispatching multiple kernels (specialized, unmodified or a combination of both) on the same machine without the need for virtualization.

2. Build DKut (Section 4.4.1) and SKut (Section 4.4.2), tools for specializing a kernel at fine (instruction-level) and coarse (system-call level) granularity, respectively.

3. Presents an evaluation for attack surface reduction and performance overheads; the proposed framework shows virtually no runtime overhead while significantly reducing the attack surface both in terms of the amount of eliminated code and CVEs.

I also discuss limitations of MultiK and also how other techniques can be adapted to integrate with it (including a comparative evaluation) in Section 4.6 and Section 4.7.

[3]In fact, MultiK is not beholden to these tools. System designers may use their favorite profiling methods/tools. The kernel profiles that are obtained can be easily integrated with the MultiK framework.

[4]Though this can be generalized as explained in Section 4.7.

[5]https://www.docker.com

4.2 BACKGROUND AND DESIGN GOALS

Kernel specialization for attack surface reduction has been studied in multiple previous works. Those prior works all aimed to identify unused parts of a bloated kernel and remove the identified parts either statically or disable them at runtime. Debloating kernels in this way removes software vulnerabilities present in unused kernel code, reducing its attacks surface and thereby contributing to a system's security. In the following, I address limitations of existing kernel attack surface reduction approaches and then set my threat model and design goals. At the end, I illustrate an overview of my approach in MultiK to achieve my design goals.

4.2.1 Limitations of existing approaches

Complexity in handling Kconfig One of the prominent approaches to de-bloating Linux kernel is to use the kernel build configuration system, *kconfig*. However, working with those configurations is a complex job not only because there are over 300 feature groups and more than 20,000 individual configuration options to choose from, but also because most of these options have many dependencies that further contribute to the complexity of the system. While approaches to automate *kconfig* to tailor the Linux kernel have been proposed [3, 66], they often require (manual) maintenance of whitelist and blacklist configuration options — these lists quickly become irrelevant as applications and kernel evolve.

Specializing a kernel for running many applications When running multiple application on a system, those approaches [3, 66] tailor the kernel for the *combined* set of applications. This negatively affects the attack surface reduction because the kernel will need to contain code for serving requests from all such applications, and not just for serving a single application. For instance, orthogonal applications (*e.g.*, Apache and ImageMagick) only share about 20% of the kernel code for their usage. Even similar applications (*e.g.*, Apache and `vsftpd`) share about 83% of the kernel and often leave as much as 20% not shared. Hence, there is a high likelihood that unused code stays in the final kernel.

Use of virtualization and specialization granularity To address some of these concerns, the Face-Change system proposed the customization of kernels for each application [18]. However, their system is implemented with a hypervisor component, and this makes applications and their kernels run in a virtual machine (with associated performance penalties). Further, due to the use of a VMI-based approach for determining the appropriate

kernel "view", Face-Change incurs additional runtime overheads (about 40% for Apache and around $5-7\%$ for Unixbench).

KASR system [2] eliminates the performance overhead (keeps it within 1%) but still requires applications to run in virtual machines. Moreover, their kernel specialization is limited to a coarse page-level granularity that can still allow unnecessary and potentially vulnerable kernel code to remain in the system. With MultiK I aim to overcome some of these limitations as I discuss in Section 4.2.3.

Application kernel profile completeness. A precondition of using any of these kernel reduction systems is an accurate and complete profile of the kernel facilities that applications depend on. If this profile is incomplete, then a benign, uncompromised application may try to execute code in the kernel that is not part of the profile. Unfortunately, executing code that is not in the customized kernel due to an incomplete profile is indistinguishable from a compromised application trying to execute code that the original application cannot invoke. The need for a complete profile is a limitation of all kernel reduction systems [2, 3, 18, 66], including MultiK.

4.2.2 Threat Model

In MultiK, I assume the following for attackers:

- **Local, user-level execution is given without physical access to the machine** I assume my attackers are limited to launching local/remote attacks on kernel from user privilege (*i.e.*, ring 3) without having any physical access to the machine.
- **Firmware, BIOS and processor are trusted** Attacks on the kernel originating from components at levels lower than the kernel are out of scope for this paper.
- **Hardware devices attached to the machine are trusted** Similarly, DMA attacks and other attacks from hardware devices are out of scope.

This threat model covers general kernel exploit scenarios, *i.e.*, launching an attack from user-level to interfere with kernel-level execution. For instance, the followings are examples of valid attacks on MultiK:

- Privilege escalation attacks from user to kernel.
- Control-flow hijacking attacks (arbitrary code execution) in kernel.

- Information leaks (arbitrary read) from kernel to user.
- Unauthorized kernel data overwriting (arbitrary write) originating from user.

4.2.3 Design Goals

The overarching goal of MultiK is to reduce the kernel attack surface of a system by generating a minimal kernel for running a set of user applications. To achieve this goal, I aim to build a system that can do this in an efficient (*i.e.*, no runtime overhead) and transparent (*i.e.*, no application changes) manner. I elaborate on the design goals of MultiK next.

Flexible and fine-grained attack surface reduction The first design goal of MultiK is to only permit the kernel code identified as necessary in an application's profile to run when the application is running. Some prior work [3] customizes kernels at *feature* granularity, by minimizing build-time configuration options required for supporting a specific application (*e.g.*, if access to USB mass storage is not required, `CONFIG_USB_STORAGE` can be disabled to exclude code that deals with interfacing with USB devices). Sometimes such features are big, and because the feature is its minimal granularity, it often contains more code than required even if only parts of features are used by applications. Similarly, KASR [2] customizes kernels at a *page* granularity by dynamically removing the executable permission for specific kernel code pages if the code in those page are not being used by the application. However, this approach overestimates required kernel code by including a whole page (4KB) even if application requires only parts of it. In this regard, I aim to design MultiK to support granularity down to the basic block level. I note that in theory, MultiK can even support byte-level granularity, but in practice since it would not make sense on current CPU architectures to permit a subset of instructions in a basic block to execute without permitting all the instructions in the basic block to execute, I only evaluate MultiK down to the basic block granularity. Further, prior works often could only customize at a specific granularity (*e.g.*, page level for KASR [2], feature level for Tailor [3]). My goal is to design MultiK to be a framework that can orchestrate kernels specialized at different specialization granularity.

Fine-grained security domain for customization Another design goal for MultiK is to customize kernel at a fine-grained security domain level. A security domain can be a single process or an instance of a container.

Previous works customized the kernel for a whole system (*i.e.*, all applications or an application stack running on the machine) [3] or to run a whole virtual machine [2]. Such customized kernels will include the union of kernel code required by every application running on the machine. Customizing kernels for all applications together does not minimize the attack surface as each application will have more kernel code mapped than is required [18]. As a result, MultiK aims to support specialized kernels for each process so that every specialized kernel has code only necessary for the process.

Efficiency MultiK should minimize the performance overhead. Specifically, I aim to have near-zero ($< 1\%$) as shown in Section 4.6.4 run-time performance overhead and not interfere with the application execution.

Transparency MultiK should not require application source code, application code instrumentation or application changes. Further, customized kernels must be compatible with the target application and should able to support regular use-cases or workload of the application. To this end, I design MultiK to interact only with the kernel space to maximize compatibility and to support user-level applications transparently.

Flexibility in sharing system resources Some applications work and interact with each other closely (*e.g.*, Apache + git for GitLab and Apache + ImageMagick for MediaWiki). For such applications MultiK should allow applications to interact using system resources (*e.g.*, IPC, locks, *etc.*) to maximize the flexibility in application interactions. Employing virtual machine based solutions and address space isolation techniques (SFI [67], XFI [68], *etc.*), will reduce the flexibility and ease of such interactions. Ideally, MultiK should let normal interactions among applications as if they are all running on a machine with a single kernel. Running multiple applications in *Docker* containers on the same machine exemplifies this goal. In other words, I would like my design to provide customized kernels for each application (or docker container) to strike a balance between the isolation of attack surfaces and the flexibility (*i.e.*, to allow normal application interactions).

4.2.4 My Approach

MULTIK: Multi-Kernel Orchestration I present MultiK (Section 4.2), a kernel runtime that deploys tailored kernel code for each application. MultiK customizes the kernel code before launching an application by consulting the *kernel profile* of the target application. Specifically, MultiK makes (1) a copy of the entire kernel text to a new physical memory

region, (2) removes the unused parts from this kernel copy by using the application's kernel profile and (3) deploys the application with this tailored kernel. To transparently and efficiently deploy tailored kernel code for each application, page table entries for the kernel text are updated. That is, MultiK alters virtual-to-physical page mappings to point the application to the new customized kernel. By doing this, I can switch automatically the kernel between applications (with near-zero performance overhead) because the page table will be switched as part of a context switch.

MultiK can work with application *kernel profiles* produced at different granularities and using different profiling techniques. I use two specialization techniques (presented below) to demonstrate and evaluate MultiK.

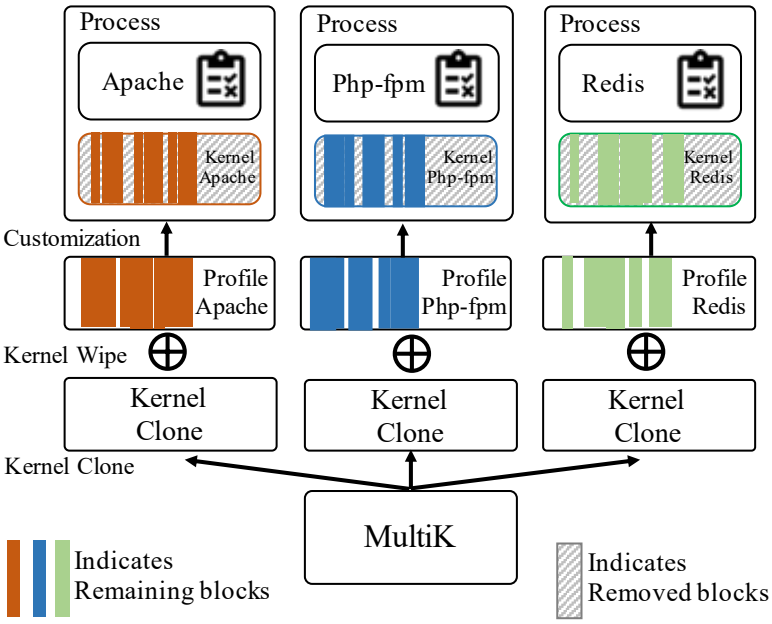


Figure 4.2: MultiK Architecture. This figure depicts the runtime kernel specialization of MultiK. When an application is launched, MultiK first fetches the application's kernel profile from the kernel profile database. Next, MultiK clones the KERNEL0 as KERNEL0' and then masks the parts that are not contained in the application's kernel profile. As a result, MultiK creates a customized kernel for the running application. MultiK applies this process for all applications running in the system, and each application runs with its own customized kernel.

D-KUT: Dynamic tracing DKut (Figure 4.6) is a tool to identify the kernel code necessary for an application. I leverage dynamic tracing of the kernel execution while the system is running the target application with its use cases. This approach has also been used in previous work (*e.g.*, [2, 3, 18, 66]). The purpose of this step is to determine which parts of

kernel code to remove before application deployment. In particular, I will remove any code that is not included in the dynamic execution traces. My tracer collects and stores executed parts of the kernel at basic-block granularity (the smallest granularity) to maximize attack surface reduction. For tracing to be effective and for the customized kernel to not impact application execution, tracing requires a good set of use cases that are representative of an application's kernel usage. Previous work (*e.g.*, [3]) has found that a modest amount of test cases and tracing runs are sufficient to get good coverage of an application's kernel profile. Here I assume that such test cases are supplied by application developers or deployers (*e.g.*, from unit tests, benchmarks, workloads *etc.*). Generating the workloads or test cases is not in scope for this work because MultiK does not limit how the kernel is profiled. Since different applications exercise different parts of the kernel, I need to carry out the tracing for each application and store them as an application's *kernel profile*. This collected kernel profile is used by MultiK when it generates a tailored kernel for an application at runtime. **Note** that MultiK works independently of the kernel customization technique and can use a kernel profile generated by any existing or future customization techniques.

S-KUT: Syscall Analysis SKut is an approach to expand the kernel profile and enhance the reliability. To be more specific, SKut tries to include rarely-triggered exception/ error handling code while being able to remove a large portion of the kernel. I use the compiler features to analyze the kernel source code. By doing this, I can have precise information of what code should be included. For this approach to be effective, the deployer has to have a list of system calls that the application uses by either static (*i.e.*, symbolic execution) or dynamic methods (*i.e.*, `strace`). MultiK does not address and limit how the list of system calls is generated.

Benefits An immediate benefit of removing large portions of unused kernel code is the resulting attack surface reduction. Vulnerabilities (both known and unknown) present in the removed code will never exist in the resulting runtime kernels. My evaluation results show that MultiK successfully removes large portions of kernel code (*e.g.*, unused kernel functions, system calls and loadable kernel modules) from the application's memory space. In many instances I achieve more than 90% reduction in kernel code, *i.e.*, `.text` (see Table 4.2 in Section 4.6.1) and consequently eliminate many vulnerabilities. For instance, MultiK eliminates 19 out of total 23 vulnerabilities (listed as CVEs) in Linux Kernel 4.4.1 when tailored for the Apache web server (see Section 4.6.1). Although MultiK does not aim to detect and eliminate specific vulnerabilities, it can reduce the system's overall risk of compromise and could serve as one layer of a *defense-in-depth* [69] strategy.

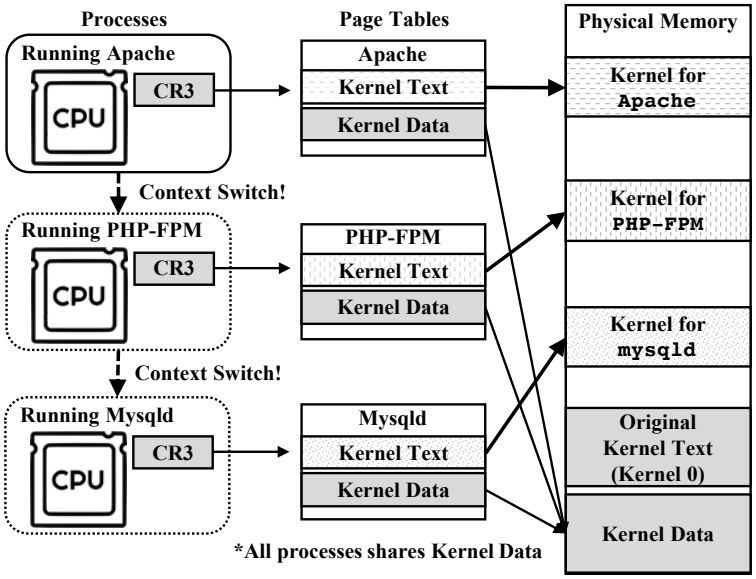


Figure 4.3: Kernel switching in MultiK. To support an efficient and transparent kernel switch, I alter each application's page table (pointed by CR3) to map the virtual addresses for kernel text to its customized kernel code. In this way, switching among customized kernels is handled automatically by regular context switch in the kernel. That is, the page table will be switched automatically by the kernel during the context switch. Note that I only customize kernel code, so kernel data is shared among all processes.

4.3 MULTIK: ORCHESTRATING SPECIALIZED KERNELS

MultiK is a kernel mechanism to orchestrate customized kernels efficiently and transparently for each application at runtime. The goal of MultiK is to provide each application access to only kernel code that is customized to that application. By doing so, MultiK will reduce the kernel attack surface by removing a large part of unused kernel code from the virtual memory of the application processes.

Overview Figure 4.3 illustrates how MultiK works with multiple applications running on the system. During the launch of each application, MultiK maps the customized kernel for the application into a new physical memory region. MultiK then updates the page table entries (of the kernel code) for that application process to redirect all further kernel execution to the customized kernel. This update to the page table will completely remove the original (full) kernel from an application's virtual memory view and switch the view to the customized one. Additionally, this update will guarantee that the CPU that runs the application can only work with the customized kernel code. This is because the process context switching in the operating system will switch the page table automatically. Hence virtual addresses in the CPU will only refer to the customized view of the kernel code.

This approach is promising because once the page table is updated during application launch (one-time cost), no runtime intervention is required to switch to the customized kernel. This will significantly enhance the runtime performance. In contrast to this approach, prior work relies on changing extended page table (EPT) permissions at runtime (in KASR [2]). This has a limitation of page-level permission deprivation, or relies on virtual machine introspection (VMI) to customize page table entries at each context switch (in FaceChange [18]). All of which incur nontrivial runtime performance overheads.

4.3.1 Challenges

Although my design is conceptually simple, MultiK must overcome a few challenges:

Sharing system resources Section 4.3.2 Running multiple applications in the same system could benefit from resource sharing among the applications. For instance, processes can communicate efficiently via inter-process communication (IPC) mechanisms available in the system such as locks, signals, pipes, message queues, shared memory, sockets, *etc.* and processes can also share hardware devices attached to the system. Such sharing is transparent by design as applications share the same kernel (*i.e.*, code) and share the same kernel memory space (*i.e.*, kernel data). However, running a customized (a different) kernel per each application could interfere with such transparent sharing. For instance, running a customized kernel in a virtual machine requires virtualizing hardware and shared resources among applications that introduces compatibility and performance issues. Additionally, having a different memory layout for each application's kernel would make the data sharing even harder. For instance, a data structure prepared in one application cannot be used in a different application and might even require transformation.

Handling hardware interrupts Section 4.3.3 A hardware interrupt can occur at any time regardless of its customized kernel view. For instance, while running a customized kernel for a non-networking application (such as `gzip`), the CPU could encounter a hardware interrupt from the network interface card (NIC). In such a case, the execution will be redirected to the interrupt handler. If the customized kernel is not equipped to handle the interrupt, then the system could either become unstable or important events could be missed. An easy workaround would be to include all hardware interrupt handlers in the customized kernel, as FaceChange [18] does. However, such an approach will increase the kernel attack surface by adding unnecessary kernel code for an application that doesn't need access to particular hardware.

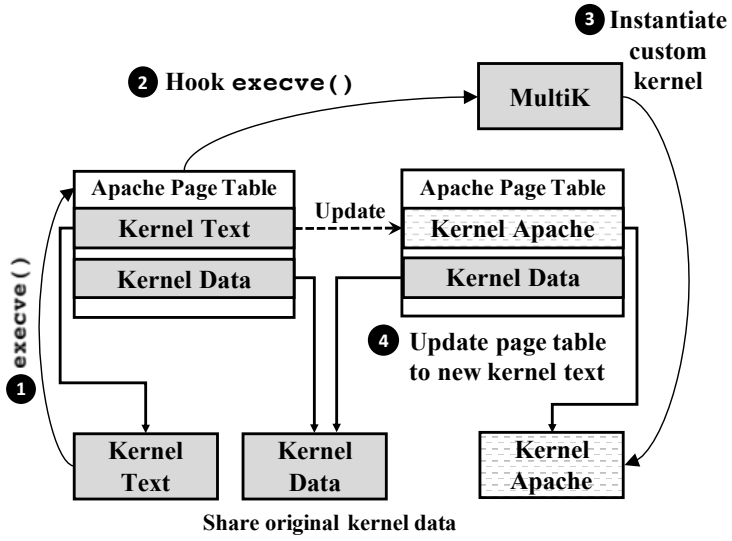


Figure 4.4: MultiK's hook in `execve`. On an application's execution, MultiK will map the customized kernel for the application and then update the application's page table entries for kernel text to point this custom kernel.

4.3.2 Deploying Customized Kernels

When deploying customized kernels in MultiK, applications should only have a view of the kernel customized for them. At the same time, I would like to ensure that customized kernels running in one system could share system resources, *e.g.*, share the memory space, available IPC mechanisms *etc.* To achieve these goals in a manner transparent to the application, I update an application's page table entirely (corresponding to kernel text) to point to the customized kernel and I do this when launching the application (*i.e.*, at `execve()`).

Deploying the customized kernel by updating kernel text page table entries gives us the following benefits. First, sharing system resource among multiple kernels becomes straight forward. Each application shares the kernel data memory space because I only alter the page table entries for kernel text. Sharing system resources via shared memory, direct memory access, or sharing kernel data structures is transparent. Second, loading and switching between kernels is simple and efficient. Loading a custom kernel will only incur the costs for mapping of new code section and updating page table entries, which is simple to do as part of the `execve` system call. Additionally, switching between the kernels is simple because a typical context switch involves changing the current page table pointer (`CR3` register in x86) to that of the newly scheduled application, and this will automatically switch the kernel text too. In short, the goal of providing applications access to only customized kernel code will be met in a transparent manner. Therefore, MultiK always isolates the application's kernel view to the one customized for it.

Figure 4.4 (from **1** to **4**) shows how MultiK switches the page table entries during application launch. To intercept the launching of an application (**1**), MultiK places a hook in the `execve` system call and maps the customized kernel in that hook (**2**). In particular, MultiK allocates a new physical memory region and copies customized kernel code to that region (**3**). Customized kernel code could either be generated on the fly using a pre-learned kernel profile for the application or could be pre-generated and stored to reduce application launch overhead. I follow the former approach since it only incurs a small one-time launch delay of $3ms$. Next, MultiK updates the application's page table entries for kernel text to point to this new physical memory region (**4**). After this point, the application can only access the customized kernel text because the original full kernel image will never exist in its virtual memory space. Then, MultiK gives control back to the `execve` system call to handle the loading and linking of the user space, and finally, I let the execution continue in the user space.

The deployment of the customized kernel and the application is finished at this stage because kernel switching for each application will be handled automatically by the regular context switching mechanism as previously discussed. For instance, Figure 4.3 shows a case of three applications, `Apache`, `Php-Fpm` and `MySQL` running in a MultiK system. When a processor (CPU) runs `Apache` in the userspace, because this application's page table maps kernel code to the kernel customized for `Apache`, the CPU can interact only with the customized kernel and cannot access the original full kernel code. When a context switch happens, say, switching to `php-fpm`, the regular context switching mechanism will change the value of `CR3` register to the page table of `php-fpm`. By design, the page table for `php-fpm` will redirect all accesses to kernel code to the kernel customized for `php-fpm`. No matter how a context switch happens in the system, MultiK ensures that an application can only have a view of kernel code that is customized for it, thus reducing the potential kernel attack surface available to any application.

Sharing system resources I design MultiK to allow sharing of system resources such as hardware devices, kernel data structures (*e.g.*, task_struct, *etc.*), inter-process communication (IPC) mechanisms (*i.e.*, pipes, UNIX domain sockets, mutexes), shared memory, *etc.*– to maximize a system's flexibility and compatibility. More specifically, I design MultiK to work with existing container mechanisms (such as Docker) and this requires the sharing of system resources while running customized kernels for each container. MultiK achieves this by *i*) fully sharing kernel data among customized kernels and *ii*) having the same memory layout for all customized kernel text.

First, kernel data and its memory space can be shared as is because MultiK does not

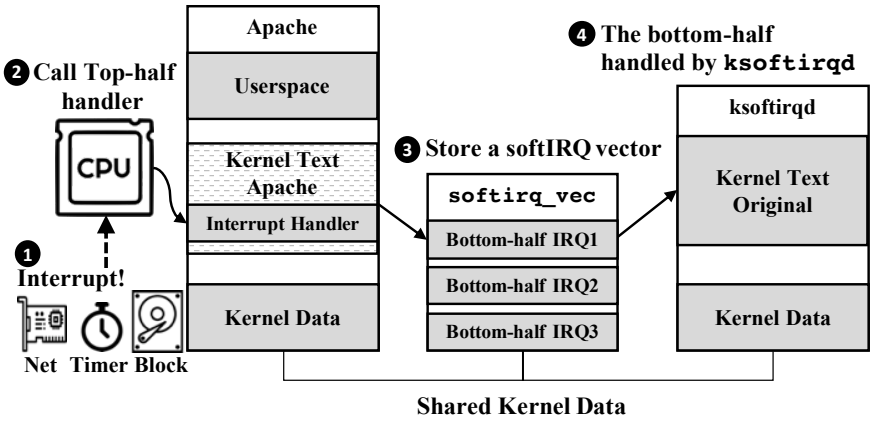


Figure 4.5: Interrupt handling in MultiK. 1) In MultiK, CPUs that run customized kernel can receive a hardware interrupt that is not related to an application's execution. 2) To handle such interrupts, I include only the part for handling the top-half of the interrupt to each customized kernel, and then 3) this handler will defer this interrupt (to `ksoftirqd` by storing a `softirq` vector. 4) Later, `ksoftirqd` will look at the vector and handle the bottom-half of the interrupt.

modify any of it. In particular, any resource sharing that does not involve any kernel text (such as passing of kernel data structures or device access via DMA) would not be affected by the deployment of MultiK. However, resource sharing that involves kernel text, for instance, a data structure that refers to the kernel functions such as function pointers (*e.g.*, file operations), would be affected if any of the customized kernel does have a different memory layout. To resolve this issue, MultiK requires customized kernels to have the same memory layout as the original kernel. That is, a customized kernel will be mapped at exactly the same virtual address space as that of the original kernel but the customized kernel text (code) will mask parts of the kernel not required by that specific application. Although this approach would create some memory overheads ($8MB$ for each customized kernel), this allows MultiK to enable sharing of function pointers among multiple customized kernels. As I see (Section 4.6.4.1) this memory overhead does not limit how many applications I can run in parallel in practice.

4.3.3 Handling Hardware Interrupts

To handle hardware interrupts I exploit deferred interrupt handling [70] to keep the customized kernels small. Hardware interrupts are problematic for customized kernels if the corresponding handler does not exist in the customized kernel. Because hardware interrupts caused by the system could be delivered at any time, even an application that does not

utilize any of the associated hardware could receive the hardware interrupt requests. Missing hardware interrupt handlers in customized kernels will cause such interrupt requests to fail and the failure to handle such interrupts could make the system unstable. One way to work around this issue is to include all interrupt handling routines in all customized kernels. However, such an approach would unnecessarily increase the potential attack surface of customized kernels.

Figure 4.5 illustrates I handle such interrupts in my framework. MultiK includes only the top-half hardware interrupt handlers that are compiled into the kernel by whiltelisting (in all customized kernels). The top-half handlers are smaller because their job is to transform a hardware interrupt request into a software interrupt request (softirq). Consequently, my customized kernels only deal with the top-half of any hardware interrupt and delegate the actual handling of interrupts to the kernel threads run by `ksoftirqd`. Hence, when hardware interrupts (*e.g.*, timer, network or block device interrupts) arrive, my customized kernel will run the top-half handler to store a softirq vector to delegate the interrupt handling to `ksoftirqd`. The bottom-half of the interrupt will then be handled by `ksoftirqd` when it runs.

KERNEL0 To handle the bottom-half of the interrupts, I run `ksoftirqd` on a general purpose regular kernel (that includes all parts that the system requires), that I refer to as KERNEL0. An example of KERNEL0 is a kernel in a distribution's package without any customization such as `linux-image-4.15.0-39-generic` in Ubuntu 18.04 LTS. This kernel not only handles hardware interrupts but also takes care of system-wide events such as booting, shutdown, *etc.* KERNEL0 also serves as a baseline template for kernel customization because it contains the entire kernel code required by the system. In MultiK, customized application kernels are generated by cloning the kernel `.text` region of KERNEL0 and masking parts of the kernel that are not needed by the application based on the application's kernel profile.

4.4 GENERATING APPLICATION KERNEL PROFILES

Application kernel profiles identify which parts of the kernel code are used by an application and which parts are not. I use 'granularity' as a unit to quantify the precision of the specialization profiles. The following granularity levels are used in the paper: (1) *Basic block level*: Basic block is a set of instructions that are always executed as a unit without any jumps or branches in between. The CPU either executes all the instructions in the basic block or executes none. This makes it one of the most precise levels of specialization [18]. (2) *Symbol level*: Kernel interfaces are exported as symbols using `EXPORT_SYMBOL()`. At

this level of granularity all the instructions that make up the interface are included in the profile even if only certain code paths of the interface are actually being used. (3) *System call level*: Given that syscalls are the main interface through which user space applications interact with the kernel, I can enumerate the syscalls that are used by the application and eliminate those that are not [71, 72]. (4) *Page level*: When a binary is loaded into memory, the (memory) smallest unit that can be referred to is a page (4KB or more). Tracing methods that track instruction that are being executed in the memory can only do so on a page level [2]. This will result in an entire page of instructions being included in the profile even if only a single instruction was run from that page. *(v) Feature level*: The kernel configuration system, `kconfig`, determines kernel features that need to be built into the kernel based on the state of certain configuration options. With this code can be included or eliminated only at a feature level, producing a much coarser and was used in [3]. MultiK is flexible enough to use application kernel profiles created using different profiling granularities in order to deploy customized kernels. I present and evaluate one trace-based (DKut) and one syscall-based approach (SKut) to demonstrate application kernel profile generation.

4.4.1 DKut: Dynamic Instruction Tracing

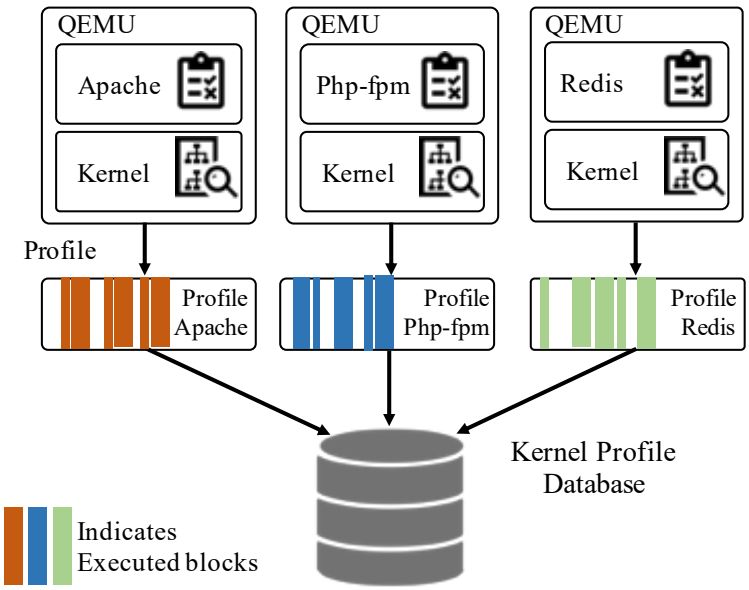


Figure 4.6: DKut Architecture. This figure depicts the offline per-application kernel tracing. While running each application with test-cases that represents application's use cases for kernel, MultiK trace the kernel and store the information for executed basic blocks during the tracing into kernel profile database.

DKut (see Figure 4.6) profiles the kernel at a basic-block level granularity. This is in contrast to KASR [2], a recently proposed approach for specializing the Linux kernel binary, that profiles the kernel usage of applications at page level ($4KB$ default page size) granularity. When profiled at page level granularity, unused instructions present in the neighborhood of used instructions within the $4KB$ range also get included in the final in-memory kernel binary.

Tracing setup. I use QEMU[73], a full system emulator to run the user application along with the vanilla kernel I need to trace. With QEMU I trace every instruction that the systems executes using the program counter (pc) register by utilizing the *exec_tb_block*. This basically helps us capture the addresses of instructions that are being *executed*.

Segmentation. The trace from above includes instructions from (1) System boot (2) Applications Runs and (3) System shutdown. The instructions from (1) and (3) are clearly not required by the application. To separate this out from the trace I use the *mmap* syscall to taint/touch a specific memory address right before the application is run and right after the application is terminated. This helps us mark the start and end of my application and helps filter out the boot/shutdown instructions from the traces.

Background processes / daemons. During a typical kernel boot it starts a bunch of background processes/daemons that provide useful services. These processes are always running with the application process. They can add noise to the traces that I obtain. To filter these out I use a custom *init* script that only starts daemons required by the application I are interested in. This eliminates noise from unnecessary background processes in the traces.

Invariants in execution. Kernel behavior is not deterministic when running an application. Execution paths in the kernel could change because not all values are invariants (*e.g.*, network conditions, CPU frequencies and time). Besides, QEMU drops trace events occasionally due to the full buffer. Therefore to capture all the possible code paths a kernel might take for a given application, I repeat the tracing process multiple times. In my experiments I observed that it takes between 10 to 15 runs before the code paths stabilize and the trace can be used with confidence as no new code paths are added for a given set of input. This need for multiple trace runs has also been reported in prior work [2, 3].

Granularity. QEMU produces a trace with *basic block* level granularity. I can post-process basic-block level traces to obtain symbol-level trace by including the entire symbol

Component	Lines of code
DKut- QEMU	3 LoC of C
DKut- Scripts	272 LoC of Python
SKut- Make	1 LoC of Makefile
SKut- Scripts	26 LoC of Python
SKut- Scripts	96 LoC of Golang
MultiK- Linux Kernel 4.4.1	477 LoC of C

Table 4.1: Source lines of code (SLOC) of the components of MultiK and DKut.

corresponding to each of the blocks. Symbol-level trace requires less number of runs because the post-processing makes the trace more inclusive. In my experiments I observe that even at this symbol granularity the trace obtained is much smaller than the ones produced by a page level trace used by KASR [2], see Table 4.2.

Compatibility with other specialization techniques. DKut only requires the final bootable kernel binary to produce a trace. Thus a kernel produced as a result of any other specialization technique can be further specialized with my tool. Thus DKut can complement and take advantage of other profiling techniques.

4.4.2 SKut: Syscall Analysis

SKut is a syscall-based analysis technique to increase the reliability of the application kernel profile. I track all possible functions that a system call can use for all such calls made by an application. This list of functions is built by analyzing the register transfer language (RTL) dumped by GCC when compiling the kernel with `-fdump-rtl-expand`. I obtain the approximate list of system calls that an application issues by using `strace` to intercept and record all system calls made in that execution context. This approach does not guarantee a complete list of system calls if some system calls are not triggered during tracing. I then expand the application profile by combining functions called by possible system calls with the original profile. In my experiments, by using this technique, I increase the coverage of the profile generated by DKut at symbol granularity. The result in Section 4.6.1 shows that more than 82% of the kernel can be reduced and 73% of the CVEs are mitigated when including functions used by system calls in the profile.

4.5 IMPLEMENTATION

I implement MultiK on Linux 4.4.1 [6] running on Intel Core i7-8086K (4.00 GHz) CPU. I use the `procfs` interface to provide an application identifier and the corresponding application kernel profile to the respective application. I generate application kernel profiles in an offline step using DKut and SKut tools (see Section 4.4). I hook the `execve` syscall so that when it is invoked to launch an application (that has a corresponding application kernel profile) I generate and deploy a specialized kernel as described in Section 4.3. Table 4.1 lists the total amount of code for MultiK. I built MultiK with 477 lines of C, DKut with 275 lines of code and SKut with 123 lines of code.

My customized kernels are masked in a special manner – by overwriting them with a special one-byte instruction sequence `0xcc` (instruction `int3`). This acts a fall-back mechanism for detecting (and reporting) when an application tries to execute code not available in its customized kernel. I choose `int3` not only because this instruction could raise a software interrupt (that my kernel can intercept to thwart unexpected execution) but also because it is a one-byte instruction whose semantics cannot be changed by arbitrary code execution resulting from attacks. Note that other techniques[18], overwrite the kernel binary with a sequence of `UD2` (two-byte) instructions, `0x0f 0x0b`. This opcode can be misinterpreted by reading it as `0x0b 0x0f`. In such cases, it would not raise an interrupt. On the other hand, using the `int3` instruction, I can detect unexpected execution in a more reliable fashion.

For instance, consider a simple case when an application's requested kernel execution hits a masked function or masked basic blocks (*e.g.*, a branch not taken or a system call not used during the tracing). In such cases, the execution will hit the `int3` instruction immediately. The kernel knows that the code pointed to (by the instruction pointer for the software interrupt) is missing. At that point, one can choose to kill the process or follow other strategies[7].

4.6 EVALUATION

I evaluated MultiK to answer the following questions:

- How much attack surface can MultiK reduce (Section 4.6.1)?
- How effective is MultiK at eliminating CVEs (Section 4.6.2)?

[6] I choose an older kernel to demonstrate MultiK's capability in reducing vulnerabilities by listing affected CVEs. Note that MultiK can be applied to the newest kernels as well to remove potential vulnerabilities.

[7] Depends on the policies picked by the system designer.

Specialized Kernel	Apache			STREAM			Perf		
	B	**S**	**SC**	**B**	**S**	**SC**	**B**	**S**	**SC**
Text (code)	93.68%	88.88%	82.25%	97.79%	95.16%	87.64%	96.33%	93.46%	86.57%
Full Symbol	91.87%	91.95%	85.78%	96.65%	96.67%	89.50%	94.78%	94.85%	88.46%
Partial Symbol	99.78%	91.95%	85.78%	99.93%	96.67%	89.50%	99.84%	94.85%	88.46%

Table 4.2: Attack Surface Reduction in comparison with the Vanilla Kernel. Text refers to eliminated kernel executable text. Full and partial symbol refer to kernel functions that are fully and partially eliminated respectively. **B**, **S** refer to kernel profiles generated by DKut with basic-block and symbol granularities respectively. **SC** refers to kernel profiles generated by SKut.

- How long does it take to generate a kernel profile for an application (Section 4.6.3)?
- What is the effect of MultiK on the system's runtime performance (Section 4.6.4)?

Evaluation setup I specialize application kernels at three different granularity: (1) block and (2) symbol (both with DKut) and (3) syscall (with SKut) (Section 4.4). I refer block, symbol and syscall granularity to **B**, **S**, and **SC**, respectively, in the rest of the paper. I performed all experiments in a KVM virtual machine with 2 vCPUs and 8GB RAM running on Intel(R) Core(TM) i7-8086K CPU @ 4.00GHz. Note that I use KVM for the convenience of testing, and MultiK does not requires a virtual machine to run specialized kernels. For specializing kernels to applications, I choose Apache, STREAM, perf, Redis, and Gzip.

4.6.1 Attack Surface Reduction

I first tackle the question of how much kernel attack surface MultiK can reduce. I do this by measuring how much kernel text (code) is reduced by specialization. Table 4.2 shows the percentage of kernel text reduction w.r.t. the vanilla kernel, and **B**, **S**, and **SC** indicate the granularity of specialization.

The first row shows the percentage of reduced kernel code. The reduction for each application is depending on the application's kernel usage. With the block level granularity, MultiK can reduce 93.68% of the kernel text from Apache (I/O intensive) and 97.79% of the kernel text from STREAM

More than 82% of the kernel text can still be removed even when the granularity is coarse such as symbol and syscall. My work outperforms KASR [2] and Tailor [3], which can reduce 64% and 54% of kernel code respectively. The second and third rows present the fully and partially removed kernel functions respectively. Because a block is smaller than a function body, I remove parts of functions. With block granularity, more then 99% of the text is

CVE	Description	Effect	Apache			STREAM			perf		
			S	B	SC	S	B	SC	S	B	SC
CVE-2018-11508	An information leak in `compat_get_timex()`	**Leak**	V	V	V	V	V	V	V	V	V
CVE-2018-10881	An out-of-bound access in `ext4_get_group_info()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2018-10880	A stack-out-of-bounds write in `ext4_update_inline_data()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2018-10879	A use-after-free bug in `ext4_xattr_set_entry()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2018-10675	A use-after-free bug in `do_get_mempolicy()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2018-7480	A double-free bug in `blkcg_init_queue()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2018-6927	An integer overflow bug in `futex_requeue()`	**DoS**	X	X	X	V	V	X	V	V	V
CVE-2018-1120	A flaw in `proc_pid_cmdline_read()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-18270	A flaw in `key_alloc()`	**DoS**	V	V	X	V	V	X	V	V	X
CVE-2017-18255	An integer overflow in `perf_cpu_time_max_percent_handler()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-18208	A flaw in `madvise_willneed()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-18203	A race condition between `dm_get_from_kobject()` and `__dm_destory()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-18174	A double free in `pinctl_unregister()` called by `amd_gpi_remove()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-18079	A null pointer deference in `i8042_interrupt()`, `i8042_start()`, and `i8042_stop()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-17807	Lack of permission check in `request_key_and_link()` and `construct_get_dest_keyring()`	**Priv**	V	V	X	V	V	X	V	V	X
CVE-2017-17806	Lack of validation in `hmac_create()` and `shash_no_set_key()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-17053	A use-after-free bug in `init_new_context()`	**DoS**	X	X	X	V	V	X	X	X	X
CVE-2017-17052	A use-after-free bug in `mm_init()`	**DoS**	X	X	X	V	V	X	X	X	X
CVE-2017-15129	A use-after-free bug in `get_net_ns_by_id()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2017-2618	Lack of input check in `selinux_setprocattr()`	**DoS**	V	V	V	V	V	V	V	V	V
CVE-2016-0723	A use-after-free in `tty_ioctl()`	**DoS**	X	X	X	X	X	X	X	X	X
CVE-2015-8709	A flaw in `ptrace_has_cap()` and `__ptrace_may_access()`	**Priv**	P	P	P	V	V	V	V	V	V
CVE-2015-5327	A out-of-bound access in `x509_decode_time()`	**DoS**	V	V	V	V	V	V	V	V	V

Table 4.3: List of CVEs tested in my security evaluation. Category numbers are referring to: **V**: the entire vulnerability is removed, **P**: parts of vulnerability is removed, **X**: vulnerability still exists. Effects are referring to: **DoS**: denial-of-service, **Leak**: information leak, **Priv**: privilege escalation. **B**, **S** refer to kernel profiles generated by DKut with basic-block and symbol granularities respectively. **SC** refers to kernel profiles generated by SKut.

excluded.

I observed that the implementation of *all* system calls in the Linux kernel [8] only takes up 14% of the text. Restricting access to system calls (*e.g.*, AppArmor [71], seccomp-bpf [74]) would (1) not remove any code (leaving the kernels vulnerable) and (2) have a lesser impact than the techniques discussed in this paper. This shows the limitations of only focusing on the whitelisting of system calls.

[8] I count all functions called by system call entry functions.

```
static int
ptrace_has_cap(struct user_namespace *ns, unsigned int mode) {
    //...
    return has_ns_capability(current, ns, CAP_SYS_PTRACE);
}
static int
__ptrace_may_access(struct task_struct *task, unsigned int mode) {
    //...
    if (ptrace_has_cap(tcred->user_ns, mode))
    goto ok;
    //...
}
```

Figure 4.7: In *CVE-2015-8709*, `__ptrace_may_access()` calls `ptrace_has_cap()` to check if the process has the capability to access an user namespace. The code is simplified.

4.6.2 CVE Case-Study

I analyze all CVEs present in Linux 4.4.1 by looking at the patch for each one and detecting which functions are vulnerable. The kernel is compiled with configuration 4.4.0-87-generic. A number of vulnerabilities are excluded because they are not present in the kernel binary since they target loadable kernel modules and I do not load modules. I find that 23 of 72 CVEs exist in the kernel binary. A CVE might involve multiple functions.

I separate the results into three categories to indicate the different levels of mitigation: (1) **V** refers to the case where all functions associated with a vulnerability are removed (2) **P** refers to case where some functions associated with a vulnerability are removed, and (3) **X** refers to case where no functions associated with a vulnerability are removed. Table 4.3 shows the result of each of the CVEs. On average, 20.3 (out of 23) CVEs are mitigated for both block and symbol granularity on average and 17 out of 23 CVEs are mitigated for the system-call granularity.

If a CVE is located in a popular code path, it is more likely that an application exercises it during the offline profiling phrase. Therefore, such CVEs (*i.e.*, *CVE-2017-17053* and *CVE-2016-0723*) are likely to remain for all applications. In other words, if a CVE (*e.g.*, *CVE-2018-11508* and *CVE-2017-15129*) is on a unpopular code path, there is a good chance to remove it. Figure 4.7 is an example of a CVE that I remove the vulnerability by partically removing one of the function required to form an exploit chain. The vulnerability is about that attackers can create a malicious user namespace and wait for a root process to enter `ptrace_has_cap()` to gain the root privilege. To exploit the CVE, the attackers need two functions, `ptrace_has_cap()` and `__ptrace_may_access()`. Hence, removing one of these functions can mitigate this CVE.

Benchmarks	Symbol	Block
Apache	1714.26	2492
STREAM	146.77	411.98
perf	2139.22	2965.15

Table 4.4: Time to complete tracing in seconds. **Symbol** and **Block** refers to DKut granularity, symbol and basic-block respectively.

I elaborate each category using a CVE for the Apache web server as an example with **B**(Block) granularity.

1. **V**, the vulnerability is entirely removed. *CVE-2017-17807* is a vulnerability resulting from an omitted access-control check when adding a key to the current task's default request-keyring via the system call, `request_key()`. `construct_key_and_link()` and `construct_get_dest_keyring()`, are required to realize the vulnerability. Therefore, since both are eliminated, attackers have no way to form the chain in order to exploit this CVE.

2. **P**, the vulnerability chain is partially removed. *CVE-2015-8709*, depicted in Figure 4.7, is a flaw that can allow for a privilege escalation. Invoking both functions form the exploit chain: `ptrace_has_cap()` and `__ptrace_may_access()`. Because my kenrel specialization partially removes one of the functions (`ptrace_has_cap()`), Attackers will no longer be able to exploit this CVE.

3. **X**, the vulnerability chain remains. *CVE-2017-17052* allows attackers to achieve a 'use-after-free' because the function `mm_init()` does not null out the member `->exe_file` of a new process's `mm_struct`. Attackers can exploit this as none of the functions have been removed.

4.6.3 Offline Profile Generation Performance

I trace applications for 10 iterations for symbol granularity and 15 for block granularity because I observed that the workload tended to be stable after this many of iterations. The profiling time depends on how long the workload runs. Table 4.4 shows the time needed to profile each of the applications. The time needed for profiling depends the workload. If the workload (STREAM[64] in Table 4.4) only takes a short amount of time to finish, the profiling will be quick and vice versa (Apache and perf in Table 4.4).

4.6.4 Performance Evaluation

In this section I evaluate (1) application performance by the Apache web server benchmark, (2) context-switches by perf[9] and (3) memory bandwidth with STREAM[64]. All the experiments were performed in a KVM virtual machine with 2 vcpus and 8G RAM running on Intel(R) Core(TM) i7-8086K CPU @ 4.00GHz. Positive % indicates improvements, and negative % indicates degradation of the application performance. Positive % indicates improvements, and negative % indicates degradation. I evaluate the performance with application benchmarks such as Apache web server, STREAM [64], a memory microbenchmark, and *perf*, a scheduling microbenchmark.

Apache Web Server I ran the Apache web server, version 2.4.25, on a specialized kernel and the client program on KERNEL0. The client program Apache benchmark sends 100,000 requests in total with 100 clients concurrently. Table 4.5 shows that the Apache web server running on specialized kernels (regardless of the trace granularity) has a very similar performance, in terms of number of requests served per second, compared to when running on a vanilla kernel. The performance is within 0.6% of the baseline.

STREAM I evaluate the memory performance with STREAM, version 5.10. STREAM has four metrics including `copy`, `scale`, `add` and `triad`. They refer to the corresponding kernel vector operations. `Copy` refers to `a[i] = b[i]`. `Scale` refers to `a[i] = const*b[i]`. `Add` refers to `a[i] = b[i]+c[i]`. `Triad` refers to `a[i] = b[i]+const*c[i]`. `Copy` and `scale` take two memory accesses while `add` and `triad` take three. Table 4.6 shows that STREAM running on specialized kernels regardless of the granularity have close performance compared to the baseline and the difference is less than 0.5% for all operations.

[9]I run the perf command: `perf bench sched all`.

Benchmarks	**Req/Sec**	
Apache-Vanilla	23401.07	0.00 %
Apache-B	23337.53	-0.27%
Apache-S	23445.03	+0.19%
Apache-SC	23536.31	+0.58%

Table 4.5: Processed HTTP requests per second when running Apache web server with and without (vanilla) MultiK. **B**, **S** refer to kernel profiles generated by DKut with basic-block and symbol granularities respectively. **SC** refers to kernel profiles generated by SKut. Green cells → better performance (more requests per second). Red cells → worse performance (less requests per second).

Perf I evaluate context switch overheads with the perf scheduling benchmark that is composed of a messaging microbenchmark and a piping microbenchmark. The messaging microbenchmark has 200 senders that dispatch messages through sockets to 200 receivers concurrently. The Piping microbenchmark has 1 sender and 1 receiver processes executing 1,000,000 pipe operations. Table 4.7 shows that perf running on specialized kernels (regardless of the trace granularity) takes the same amount of time to complete the message and pipe tasks when compared to running on vanilla (unmodified) kernels. The performance difference for both message and pipe tasks is less than 0.6%.

4.6.4.1 Memory Effect

Every specialized kernel takes up approximately $8MB$ of additional memory space. Kernel memory is not swappable. Therefore, the system-wide memory pressure will increase if I keep creating new specialized kernels. I evaluate the memory pressure by measuring the memory bandwidth with STREAM [64] benchmark. Under a higher memory pressure, the system will swap memory pages in and out more frequently and results in a lower memory bandwidth. I run STREAM together with multiple specialized kernels on a KVM virtual machine with 2 vcpus and 8GB running on Intel(R)Core(TM) i7-8086K CPU @ 4.00GHz. *Swappiness* is a kernel parameter ranging from 0 to 100, which controls the degree of swapping. The higher the value is, the more frequently virtual memory swaps. I conduct the experiment with default value, 60. In order to exclude the factor of the CPU (*i.e.*, busy CPUs cause slower memory bandwidth), I run the command `sleep` on specialized kernels. Figure 4.8 shows that memory operations start to slow down due to more frequent memory swaps when there are more than *750 coexisting kernels*. To be more specific, operations `Add` and `Triad` take twice the memory accesses than the others so the time that they need to finish

Operation	**Copy**		**Scale**		**Add**		**Triad**	
STREAM-Van.	0.008192	0.00%	0.008632	0.00%	0.011207	0.00%	0.011428	0.00%
STREAM-B	0.008201	-0.11%	0.008623	+0.10%	0.011237	-0.27%	0.011445	-0.15%
STREAM-S	0.008199	-0.09%	0.008628	+0.05%	0.011201	+0.05%	0.011418	+0.09%
STREAM-SC	0.008178	+0.17%	0.008597	+0.41%	0.011179	+0.25%	0.011406	+0.19%

Table 4.6: Average time to complete different kernel vector operations in seconds. Copy refers to `a[i] = b[i]`. Scale refers to `a[i] = const*b[i]`. Add refers to `a[i] = b[i]+c[i]`. Triad refers to `a[i] = b[i]+const*c[i]`. **B**, **S** refer to kernel profiles generated by DKut with basic-block and symbol granularities respectively. **SC** refers to kernel profiles generated by SKut. Green cells → better performance (shorter time to finish an operation). Red cells → worse performance (shorter time to finish an operation).

Benchmark	Message		Pipe	
perf-Vanilla	0.190	0.00%	10.37	0.00%
perf-B	0.191	-0.53%	10.36	+0.10%
perf-S	0.189	+0.53%	10.35	+0.19%
perf-SC	0.190	0.00%	10.38	-0.10%

Table 4.7: Time to complete each microbenchmark in seconds. Message refers to concurrent sender/receiver microbenchmark. Pipe refers to sequential piping between two process. **B**, **S** refer to kernel profiles generated by DKut with basic-block and symbol granularities respectively. **SC** refers to kernel profiles generated by SKut.

is twice longer. I also evaluated the memory effect on a machine with 4G RAM running on Intel(R) Xeon(R) CPU E3-1270 v6 @ 3.80GHz. In this case the impact is observed after 345 coexisting specialized kernels. The results indicate that memory overhead is not an issue for most practical deployments.

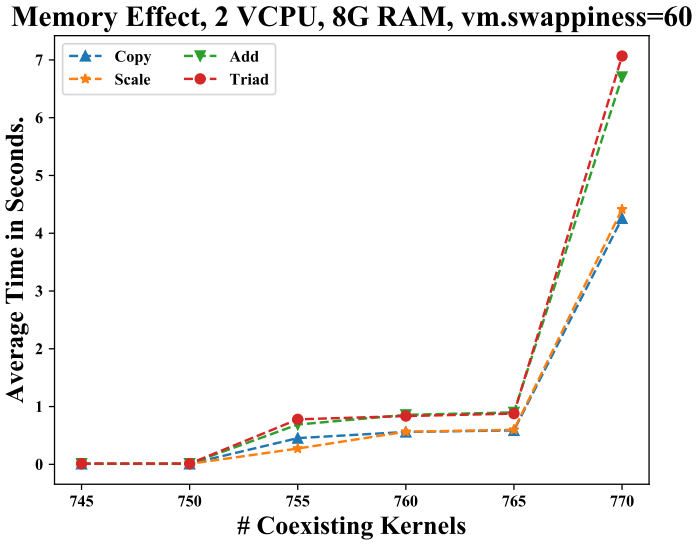


Figure 4.8: STREAM benchmark with multiple coexisting kernels.

4.7 DISCUSSION

Kernel Specialization for Containers The method of MultiK for dispatching multiple kernels can seamlessly assign and run a specialized kernel for each security domain, *i.e.*, container, such as Docker. This is done by sharing a specialized kernel for multiple applications. In particular, I can configure MultiK to have one specialized kernel for set of applications (running in a container) by profiling kernel traces while running all such application to-

gether. In my experiment, I integrated MultiK with Docker containers trivially, and show that MultiK affects Docker containers' performance by less than 1%. This approach can also reduce the memory usage by using fewer kernels. It aligns well with cloud deployment patterns where *containers* from different organizations may share the same hardware.

Kernel Page-Table Isolation Kernel page-table isolation (KPTI) [75] is a feature that mitigates the Meltdown [55] security vulnerability. KPTI uses two page tables for user and kernel modes respectively so that user-mode applications can only access a limited range of kernel memory space such as entry/exit functions and interrupt descriptors. Although KPTI hides the kernel from the user space, it does not mitigate the vulnerabilities because attackers can still call system calls with carefully crafted parameters to enter the kernel mode and exploit the system.

Kernel CFI MultiK is orthogonal to other kernel attack surface reduction techniques such as control-flow integrity (CFI) [76] and can work with them concurrently. Kernel CFI can indirectly achieve attack surface reductions by restricting available call/jump targets from a large number of control-flow transfer points (that would otherwise serve as attack surfaces). Such an approach is orthogonal to MultiK and, in fact, they can both complement each other. In particular, MultiK can run a kernel reinforced with KCFI, and MultiK can further trim such a kernel to achieve an overall better attack surface reduction. The reason that MultiK can be seamlessly combined with such techniques is due to the fact that it incurs extremely low performance overheads.

AppArmor and SELinux AppArmor [71] and SELinux [10] are Linux security modules [72] which try to achieve Mandatory Access Control. In particular, after identifying the necessary resources and capabilities, both approaches apply a profile to enable/disable them via white/blacklisting. The drawbacks of AppArmor and SELinux (compared to MultiK) is that they only remove access to syscalls that are an entry point to a certain code path by limiting the POSIX capabilities. The code is still available to the attacker if she can bypass [11] this protection. In MultiK I explicitly remove code paths that are not required by the application, thus preventing the attacker from accessing it altogether even if other security measures are compromised. In addition to that, MultiK can further reduce code within a system call if I apply smaller specialization granularity than a symbol, *e.g.*, basic-block granularity.

[10] `https://www.nsa.gov/What-I-Do/Research/SELinux`
[11] `https://www.cvedetails.com/cve/CVE-2017-6507`

Arbitrary Kernel Execution If an attacker is able to execute arbitrary code within the kernel space, *e.g.*, by inserting kernel modules, then the attacker can modify the page tables for applications and bypass the kernel view imposed by MultiK. I prohibit kernel module insertion for specialized kernels. If a kernel module is needed, it can be inserted from KERNEL0 and it is visible to all specialized kernels.

Kernel Data-Only Attacks As the underlying kernel data structures are shared among all the multiplexed customized kernels, MultiK alone cannot prevent kernel data corruption attacks (*e.g.*, [77, 78, 79, 80]) although it can make it harder for attackers to exploit. However it can be integrated with existing kernel data protection mechanisms (*e.g.*, [78]) to improve the overall security of the system.

4.8 CONCLUSION

MultiK does not aim to provide the same level of isolation as virtual machines(VM). Instead, it is a framework that runs specialized kernel code without losing the flexibility of integrating existing security mechanism to defend against different cyber-threats *e.g.*, data-only attacks. My evaluation shows that MultiK can effectively reduce the kernel attack surface while multiplexing multiple commodity/specialized kernels with less than 1% overhead. MultiK can be easily integrated with container-based software deployment to achieve per-container kernels with no changes to the application.

CHAPTER 5: PRACTICAL AND EFFECTIVE DEBLOATING FRAMEWORK

5.1 OVERVIEW

This chapter presents Cozart, the practical and effective framework. I learn that removing kernel parts without regarding the existing workflow of kernel components (*e.g.*, removing parts of a module or function) leads unstable kernels. To make kernel debloating practical, stability is a must. The de facto method to debloat kernels in a stable way is *feature-driven kernel debloating*. feature-driven debloating respects the kernel design and remove unnecessary code on a feature basis. If a part of a feature is necessary, the entire feature will be included in the debloated kernel. This chapter focus on feature-driven debloating on Linux. Linux is highly configurable as it exposes thousands of configuration options to configure its features (*e.g.*, file systems and device drives).

I build and evaluate a framework named Cozart that automatically debloats kernels based on configuration options. I use Cozart as a vehicle to study the limitations of practical kernel debloating and present the insights to shed light on addressing these limitations.

5.2 INTRODUCTION

5.2.1 Motivation

Commodity operating systems (OSes), such as Linux and FreeBSD, have grown in complexity and size over the years. However, each application usually requires a small subset of OS features [81, 82], rendering the OS kernel bloated. The bloated OS kernel leads to increased attack surface, prolonged boot time and increased memory usage. Application-oriented OS kernel debloating—*reducing the kernel code that is not needed by the target applications*—is reported to be effective in mitigating many of the above issues. For example, Kurmus *et al.* [3] report that 50%–85% of the attack surface can be reduced by debloating the Linux kernel for server software. Alharthi *et al.* [17] show that 34%–74% of known security vulnerabilities can be nullified in the Linux kernel if it only includes kernel modules that are needed by the target application(s).

Recent trends in Function-as-a-Service (FaaS) and microservices, where numerous kernels are often packed and run in virtual machines (VMs), further highlight the importance of kernel debloating. In these scenarios, most VMs run small applications [10, 21, 83, 84], and each application tends to be "micro" with a small kernel footprint [85, 86, 87]. Some of the

recent virtualization systems, such as LightVM [21], require users to provide *minimalistic* Linux kernels specialized for target applications.

Debloating OS kernels by hand-picking kernel features is impractical. The complexity of commodity-off-the-shelf (COTS) OSes (e.g., Linux) makes manual debloating infeasible [3, 88, 89]. Linux 4.14, for example, has more than 14,000 configuration options with hundreds of new configuration options introduced every year (see Figure 5.2). Kernel configurators (such as KConfig [90]) do not help debloat the kernel but only provide a user interface for selecting configuration options. Given the poor usability and incomplete and even incorrect documentation [89], it is prohibitively difficult for users to select the minimal kernel configuration manually.

Recently, several *automated* kernel debloating techniques and tools have been proposed and built [3, 12, 21, 39, 51, 66, 91, 92, 93, 94, 95]. Despite their different implementations, existing kernel debloating techniques share the same working principle with the following three steps: (1) running the target application workloads, meanwhile tracing the kernel code that were executed during the application runs; (2) analyzing the traces and determining the kernel code needed by the target applications; (3) building a debloated kernel that only includes the code required by the applications.

Configuration-driven (also known as feature-driven) kernel debloating techniques are the *de facto* method for OS kernel debloating [3, 21, 39, 51, 66, 91, 92, 93, 94, 95]. Most existing kernel debloating tools use configuration-driven techniques as they are among the few techniques that can produce stable kernels. Configuration-driven kernel debloating reduces kernel code based on *features*—a kernel configuration option is corresponding to a kernel feature. A debloated kernel *only* includes features for supporting the target application workloads. As shown by recent studies, configuration-driven kernel debloating can effectively reduce the size of the kernel by 50%–85% [3], attack surface by 50%—85% [3], and security vulnerabilities by 34%–74% [17]. This paper focuses on configuration-driven kernel debloating and their open-source implementations.

However, despite their attractive benefits regarding security and performance, automated kernel debloating techniques are seldom adopted in practice, especially in production systems. This is certainly not due to a lack of demand—I observe that many cloud vendors (e.g., Amazon AWS, Microsoft Azure and Google Cloud) reduce the Linux kernel code in a handcrafted manner, which is not as effective as automated kernel debloating techniques (Section 5.4). The goal of this paper is to understand the limitations of kernel debloating techniques that hinder their practical adoption and to share my experience to shed light on addressing the limitations.

5.2.2 Accidental and Essential Limitations

I identify five major limitations of existing kernel debloating techniques based on my own experiences of using existing kernel debloating tools and my interactions with practitioners.

- *No visibility of kernel boot phases.* Existing debloating techniques use `ftrace` that can only be initiated after the kernel boot and hence cannot observe what kernel code is loaded during the boot phase[1]. As a consequence, critical modules (*e.g.*, disk drivers) might be missing, and the debloated kernel generated by existing techniques oftentimes cannot boot [91]. My experiments show that up to 79% of kernel features can only be captured by observing the boot phase. Even if the debloated kernel can boot, the kernel could miss certain performance and security features loaded at the boot time, leading to reduced performance and security. I find that many performance and security features are only loaded at the boot time, e.g., `CONFIG_SCHED_MC` for multicore support and `CONFIG_SECURITY_NETWORK`.

- *Lack of support for fast application deployment.* Using existing tools, debloating for a new combination of applications or deploying a new application on a debloated kernel would require going through the entire three-step debloating process of tracing, analysis, and building as described in Section 5.2.1. The process is time-consuming (can take hours or even days) and thus prevent agile application deployment—starting up a new application would have to wait until the generation of the debloated kernel.

- *Coarse-grained tracing.* Existing techniques use `ftrace`, which can only trace kernel code at the function level. However, function-level tracing is too coarse a granularity to track configuration options that affect intra-function code.

- *Difficulties in covering the complete kernel footprint.* Since kernel debloating uses dynamic tracing, it requires application workloads (typically encoded in benchmarks) to drive kernel code execution. However, it is challenging for benchmarks to cover the complete kernel footprint of the target applications, and the debloated kernel would crash if the application reaches kernel code that was not observed during tracing.

- *Not distinguishing what is executed versus what is needed.* Existing techniques treat any kernel features that are reached during the application workloads, even though the executed code may not be actually needed. For example, a second filesystem might be initialized but never needed.

[1] I show `ftrace`'s incapability of observing the boot phase even with the RAM disk based solution (`initrd`) in Section 5.8.1.

To analyze the above limitations, I divide them into *essence* (the limitations inherent in the nature and assumptions of kernel debloating) and *accidents* (those that exist in current kernel debloating techniques but that are not inherent). I believe that the first three limitations are accidental and can be addressed by improving the design and implementation of the debloating techniques, while the last two limitations are essential that require efforts beyond the debloating techniques themselves.

5.2.3 My Study and Results

This paper studies the limitations of existing OS kernel debloating techniques towards making OS kernel debloating practical, effective and fast. I focus on OS kernels deployed in cloud environments that are commonly used as guest OSes to run untrusted applications often with stringent performance requirements.

To study the limitations, I carefully designed and implemented an advanced OS kernel debloating framework named Cozart[2] built on top of QEMU [73]. Cozart employs a low-level tracing provided by QEMU to obtain visibility of the boot phase. Cozart uses instruction-level tracing to track the kernel code and map the kernel code to kernel configuration options. Cozart moves kernel tracing off the critical path: with Cozart, one can generate Configlets (a set of configuration options) specific to an application or a deployment environment *development*. I term the former Applets and the latter Baselets. Given a set of target applications, Cozart can generate a debloated kernel by directly *composing* the Applets and the Baselet generated development into a full kernel configuration. Such composability enables Cozart to *incrementally* build new kernels by reusing Configlets (saving repetitive tracing efforts) and previous build files (e.g., the object files and LKMs).

I use Cozart as a vehicle to conduct a number of experimental studies on different types of OS kernels (including Linux, the L4 microkernel and Amazon Firecracker) with a wide variety of applications (including HTTPD, Memcached, MySQL, NGINX, PHP and Redis). My studies lead to the following findings and results:

- Existing kernel debloating techniques initialize in-kernel tracing methods (*e.g.*, ftrace) too late and cannot observe the boot phase, which is critical to producing a bootable kernel. Cozart leverages the tracing feature provided by the hypervisor to support end-to-end observability (including the complete boot phase and application workloads) and produce stable kernels that can always boot and run applications stably. The debloated kernel by Cozart does not have performance regressions (but slightly

[2]Cozart is publicly available at `https://github.com/hckuo/Cozart`.

improves application performance) while achieving more than 13% reduction of boot time and more than 4MB memory savings.

- Kernel debloating can be done within tens of seconds if the configuration options of target applications are known. The composability of Cozart allows users to prepare Applets during the development phase. To deploy a new application on a debloated kernel, Cozart composes the Applet of the target application with the Configlets of the current kernel to generate a new debloated kernel within tens of seconds.

- Using instruction-level tracing (instead of `ftrace`) can address kernel configuration options that control intra-function features. The overhead of instruction-level tracing is acceptable for running test suites and performance benchmarks, given that those can be done during the development phase.

- An essential limitation of using dynamic-tracing based techniques for kernel debloating (which are the *de facto* approach) is the imperfect test suites and benchmarks. Specifically, the official test suites of many open-source applications have low code coverage. Combining different workloads (e.g., using both test suites and performance benchmarks) to drive the application could alleviate the limitation to a certain extent.

- Domain-specific information can be used to further debloat the kernel by removing the kernel modules that were executed in the baseline kernel but are not needed by the actual deployment. Take Xen and KVM as examples. I can use Cozart to further reduce the kernel size by 40+% and 39+% based on the `xenconfig` and `kvmconfig` configuration templates provided by the Linux kernel source.

- Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and extensively customized kernels (e.g., the Firecracker kernel). The reduction is significant: 47.0% for the L4 microkernel and 19.76% for the Firecracker kernel.

5.3 KERNEL CONFIGURATION

This section provides an overview of kernel configuration ecosystem using Linux as an example. Kernel debloating is not specific to Linux, and Cozart is generic and portable to different OSes.

```c
static int send_signal(...) {
  int from_ancestor_ns = 0;
#ifdef CONFIG_PID_NS
  from_ancestor_ns = ...;
#endif
  return __send_signal(...,
         from_ancestor_ns);
}
```

(a) Statement (C preprocessor)

```c
#ifdef CONFIG_TRANSPARENT_HUGEPAGE
struct page *vm_normal_page_pmd(...) {
  unsigned long pfn = pmd_pfn(pmd);
  ...
out:
  return pfn_to_page(pfn);
}
#endif
```

(b) Function (C preprocessor)

```
cachefiles-y := bind.o daemon.o ...

cachefiles-$(CONFIG_CACHEFILES_HISTOGRAM)
↪  += proc.o

obj-$(CONFIG_CACHEFILES) := cachefiles.o
```

(c) Object file (Makefiles)

Figure 5.1: Kernel configuration that controls different granularities of kernel code in Linux kernels. The same patterns are common in other kernels such as FreeBSD, L4, and eCos.

5.3.1 Configuration Options

A kernel configuration is composed of a set of configuration *options*. A kernel module could have multiple configuration options, each of which controls which code will be included in the final version. Configuration options control different granularities of kernel code including statements, functions as well as object files that are implemented based on C preprocessors and Makefiles entries.

C preprocessors select code blocks based on `#ifdef` or `#ifndef` directives — configuration options are used (as macros) to determine whether or not to include conditional groups in the compiled kernel. Figures 5.1a and 5.1b show examples of C preprocessors for two configuration options, with the granularity of statements and functions, respectively.

Makefiles are used to determine whether or not to include certain object files in the compiled kernel. For instance, `CONFIG_CACHEFILES_HISTOGRAM` and `CONFIG_CACHEFILES`, in Figure 5.1c, are both configuration options. The detailed characteristics of kernel configuration patterns have been studied before [96, 97, 98].

Note that statement-level configuration options (Figure 5.1a) cannot be identified by function-level tracing (*e.g.*, `ftrace`) used by existing kernel debloating tools [3, 51, 66, 95]. I find that 31% of C preprocessors in Linux 4.18 are options at the statement level.

The number of configuration options in modern monolithic OS kernels is increasing rapidly as a result of the rapid growth of the kernel code and features. Figure 5.2 shows the growth of kernel configuration options of Linux and FreeBSD on a yearly basis. Taking Linux as an

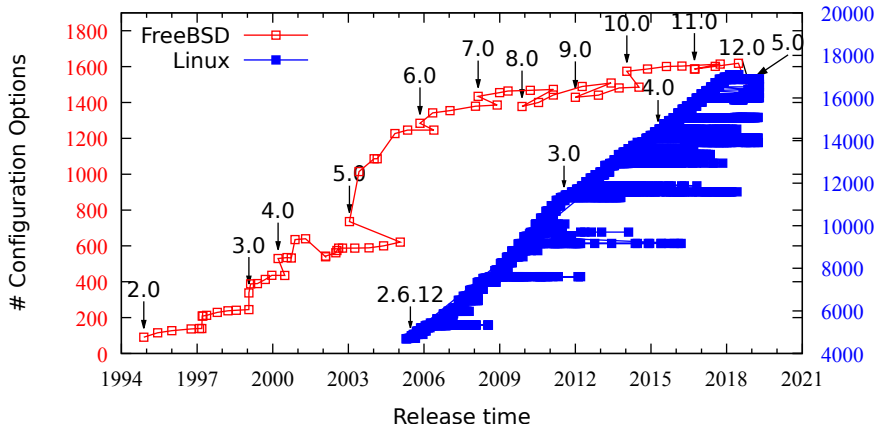


Figure 5.2: The unrelenting growth of configuration options in Linux and FreeBSD showing the need for debloating. Each point is a release tag of the code repositories.

example, the Linux kernel 5.0, 4.0 and 3.0 have 16,527, 14,406, 11,328 configuration options respectively.

5.3.2 Configuration Languages

Different OS kernels employ a variety of configuration languages to instruct the compiler on which code to include in the compiled kernel. For example, Linux and L4/Fiasco use KConfig [99], eCos uses a component definition language (CDL) [100], and FreeBSD uses a simple key-value format [101]. Despite different language syntaxes, the configuration language allows the definition of configuration options and dependencies between them. Without loss of generality, I use KConfig as an example of kernel configuration semantics through the rest of this paper.

Configuration Option Types. A configuration option in KConfig could be `bool`, `tristate` or `constant`. A `bool` option means the code will either be *statically* compiled into the kernel binary or excluded, while `tristate` allows the code to be compiled as a Loadable Kernel Module (LKM)—a standalone object that can be loaded at runtime. A `constant` option can have a string or numeric value that is provided as variables to the kernel code (*e.g.*, kernel log buffer size: `CONFIG_LOG_BUF_SHIFT`). In Linux 4.18, 48% of configuration options are `bool` (must be compiled into the kernel,*e.g.*, threads and memory), 49% are `tristate` (mostly device drivers), and the remaining 3% is `constant`.

Configuration Dependencies. Two configuration options `A` and `B` can have one of the following dependencies:

- `A depends on B`: `A` can be selected only if `B` is selected (*e.g.*, `TMPFS_XATTR` depends on `TMPFS`);
- `A selects B`: `B` is forcefully selected if `A` is selected (*e.g.*, `TMPFS_POSIX_ACL` selects `TMPFS_XATTR`);
- `A implies B`: `B` is selected by default if `A` is selected (*e.g.*, `FUTEX` implies `RT_MUTEXES`). However, implied options (`B` in the above example) can be unselected explicitly.

The Linux kernel configuration system, KConfig, employs a recursive process to enforce the above configuration dependencies [90]. It recursively selects configuration options that are `implied` and `selected` and deselects options whose `depend` relationship is not met. The final kernel configuration always follows valid dependencies but could be different from user input (if the input configuration violates dependency requirements).

5.3.3 Configuration Templates

The Linux kernel ships with a number of *manually crafted* configuration templates that includes:

- `defconfig` for the default configuration,
- `tinyconfig` includes very basic modules [25]
- `localmodconfig` disables all loadable kernel modules that are not currently loaded.
- `kvmconfig` enables options for KVM guest support.
- `xenconfig` enables options for Xen `Dom0` and guest support.

I would like to note that configuration templates are not a solution to the kernel debloating challenges because the configuration templates are all handcrafted and hardcoded. They cannot adapt to different hardware platforms and do not have knowledge of the application requirements. For example, I find that the kernel built from `tinyconfig` cannot boot on standard hardware [25], not to mention supporting applications. Some prior work [51, 52, 95] treats `localmodconfig` as a minimized configuration; however, `localmodconfig` shares the same set of limitations of static configuration templates: it does not debloat configuration options that control statement- or function-level C preprocessors and does not deal with loadable kernel modules.

`kvmconfig` and `xenconfig` are customized for kernels running on KVM and Xen setups. They provide domain knowledge of the underlying virtualization and hardware environment. In Section 5.8.5, I use the information encoded in these templates to explore the potential of debloating kernels with domain knowledge.

5.4 LINUX KERNELS IN THE CLOUD

Since I target kernel debloating for cloud-based environments, I conduct a study on existing OS kernels provided by major cloud service providers, including Amazon AWS, Google Cloud and Microsoft Azure.

In current cloud services, Linux is the dominant OS kernel in VMs provided by Amazon AWS, Microsoft Azure and Google Cloud. Most cloud vendors only provide Linux and Windows servers with Ubuntu, being the most popular distribution [102, 103]. Amazon provides its customized Linux-based server named Amazon Linux 2 [104] for EC2 instances. Therefore, I focus on Ubuntu and Amazon Linux 2 as the representative kernels in the cloud.

I analyze the Linux kernels provided by official Virtual Machine (VM) images. I find that the cloud vendors all debloat the vanilla Linux kernel to some extent. However, the debloating efforts are manual and sometimes *ad hoc*. As shown in Table 5.1, the customization by cloud vendors is often done by directly removing LKMs from the file system (or adding new LKMs). Table 5.1 shows the inconsistency between the actual number of LKM files and the number of LKMs recorded at the compilation time during kernel builds. My observations show that the kernels are initially built with many more LKMs that are then removed later from the built kernels. A drawback of manually trimming LKM binaries is the potential for violating dependencies (an LKM could depend on multiple other LKMs that may have been manually trimmed).

Importantly, there is a significant potential of improvements to further debloat the kernel based on the applications. Table 5.1 (the bottom half) compares the kernels provided by cloud vendors with the debloated variants produced by Cozart for the official Ubuntu and Amazon Linux 2, which are the baselines respectively. I use the Apache Web Server (HTTPD) as the target application.

As shown in Table 5.1, Cozart can further reduce the kernel size significantly on top of the manually customized Linux kernels. Overall, Cozart can achieve reductions of the kernels by 34.91% for Ubuntu and by up to 40.72% for Amazon Linux 2, respectively, for HTTPD as the target application. I will report the detailed results for other applications in Section 5.8.

I also studied the minimized kernel used by Amazon FireCracker, which is a micro VM specialized for Function as a Service (e.g., AWS Lambda). The Firecracker kernel is based

on Linux with handcrafted kernel configuration (910 configuration options), which offers memory overhead of less than 5MB per thread in a guest VM to allow the creation of up to 150 Firecracker instances per second per host. I can see from Table 5.1 that kernel debloating can still benefit from such a minimized kernel with a reduction of 17.69% using HTTPD as the target application.

5.5 COZART

Cozart is an advanced OS kernel debloating framework. I use Cozart as a vehicle to understand both the accidental and essential limitations of kernel debloating techniques. Specifically, Cozart integrates my solutions to the accidental limitations in its design, while including support for analyzing the essential limitations.

I chose to develop a new tool instead of building on top of existing tools (e.g., Undertaker [3] and LKTF [51]) because the new design would require substantial architectural changes and does not fit in the current implementation of existing tools.

5.5.1 Design Principles

Cozart shares the same high-level working principle as the state-of-the-art kernel debloating techniques [3, 21, 39, 66, 91, 92, 94]. It traces the kernel footprint reached by the target application workloads to determine the kernel configuration required by the target applica-

	Kernel	**# Options***	**Kernel Size**
Ubuntu	Ubuntu 18.10 Cloud Image	2495 / 5147	62007918
Debloated by Cozart for HTTPD		-51.28%/-99.86%	-65.09%
AWS	Amazon Linux 2	1214 / 946 (929)	58917847
Debloated by Cozart for HTTPD		-29.82%/ -97.46 (-97.42)%	-59.28%
AWS	Firecracker	910 / 0	15136127
Debloated by Cozart for HTTPD		-17.69% / NA	-19.61%
Google	Ubuntu 18.10 Minimal	2454 / 989 (4993)	57155890
AWS	Ubuntu 18.10 Minimal	1966 / 949 (3075)	53605858
Azure	Ubuntu 18.10	1745 / 859 (1799)	53312392

Table 5.1: Configuration options and sizes of the Linux kernels provided by Ubuntu, Google, AWS, and Azure. "Kernel size" includes kernel binary and LKMs. "# Options" is categorized into "kernel binary / LKM." Many kernels are customized by removing or adding LKMs; the actual LKMs are inconsistent with the number at the compilation time (in parentheses).

tion. The debloated kernel will only include those kernel features instead of all available features or features enabled in default configurations. Cozart distinguishes itself from the state-of-the-art kernel debloating techniques [3, 21, 39, 66, 91, 92, 93, 94] in the following aspects:

- **End-to-end visibility.** Cozart targets OS kernels used in cloud-based virtual machines. Therefore, Cozart leverages the visibility of the hypervisor to implement end-to-end observations that are able to trace both the kernel boot phase and application workloads. I build Cozart on top of QEMU.

- **Composability.** A key principle behind Cozart is to make kernel configuration *composable*. Cozart divides a kernel configuration into a set of Configlets. A Configlet is a set of configuration options used to boot the kernel on a given deployment environment (e.g., a VM) or a set of configuration options needed by a target application (e.g., HTTPD). The former is named Baselet and the latter is named Applet. Figure 5.3 shows examples of Baselet of the Firecracker MicroVM and an Applet of HTTPD. Note that a Baselet is not necessary the minimal set configuration to boot on a specific hardware. Instead, it is a set of configuration options that Cozart traces during the boot phase. Therefore, a Baselet is specialized for the hardware that Cozart traces. A Baselet can be composed with one or more Applets that are generated with the same Baselet to generate the final kernel configuration, which can be used to build the debloated kernel.

- **Reusability.** The Configlets can be stored in persistent storage and be reused as long as the deployment environment, and the application binaries do not change. Cozart moves kernel tracing and application workload runs off the critical path of the kernel debloating: with Cozart, one can conduct kernel tracing and application workload during the *development* phase and maintain the resultant Configlets in databases. The reusability also avoids repetitive tracing and workload runs, making Configlet generation a one-time effort.

- **Support for fast application deployment.** The reusability and composability form the foundation of supporting fast deployment in Cozart. Given a deployment environment (e.g., a VM) and the target application(s), Cozart can effectively retrieve the Baselet and Applets and compose them into the kernel configuration. Cozart then builds the debloated new kernel using the generated kernel configuration. Cozart also caches the intermediate build results (e.g., the object files) corresponding to the Configlets and uses the incremental build support provided by the build system.

```
CONFIG_HYPERVISOR_GUEST
CONFIG_KVM_GUEST
CONFIG_PARAVIRT
CONFIG_VIRTIO
CONFIG_VIRTIO_BLK
CONFIG_VIRTIO_MMIO
CONFIG_VIRTIO_NET
CONFIG_VIRTUALIZATION
...
```

(a) Baselet of Firecracker

```
CONFIG_EPOLL
CONFIG_EXT4_ENCRYPTION
CONFIG_EXT4_FS
CONFIG_EXT4_FS_POSIX_ACL
CONFIG_EXT4_FS_SECURITY
CONFIG_FUTEX
CONFIG_INET
CONFIG_IP_MROUTE
...
```

(b) Applet of HTTPD

Figure 5.3: Examples of Configlets.

- **Fine-grained configuration tracing.** Cozart uses Program Counter (PC) based tracing provided by QEMU to identify configuration options based on the corresponding code patterns described in Figure 5.1. I chose PC tracing based on the observation that function-level tracing (e.g., `ftrace`) cannot deal with configurations inside functions (Figure 5.1c).

The ability to generate fine-grained Configlets and the composability of Configlets also allows us to compare Applets generated by various test suites and benchmarks and to further customize Baselets based on domain knowledge.

Figure 5.4 presents the overview architecture of Cozart. Cozart requires three types of inputs: (1) the kernel source code, (2) the baseline configuration and (3) the application workloads that drive the kernel execution.

Cozart operates in the following two phases (Figure 5.4):

- **development phase.** Config Tracker is used for tracking the configuration options used by the deployment environment and target application **1**; Configlet Generator is used to generate Configlets (including both Baselets and Applets and store them in Configlet DB **2**).
- **production phase.** Given the target deployment environment and the target applications, Cozart uses Kernel Composer to generate the kernel configuration by composing the corresponding Baselet and Applets **3** and then uses Kernel Builder to build the debloated kernel **4**.

5.5.2 Tracking Configuration Options

Cozart tracks kernel configuration options used during the kernel execution driven by the target application. Cozart is expected to trace every individual application in order to

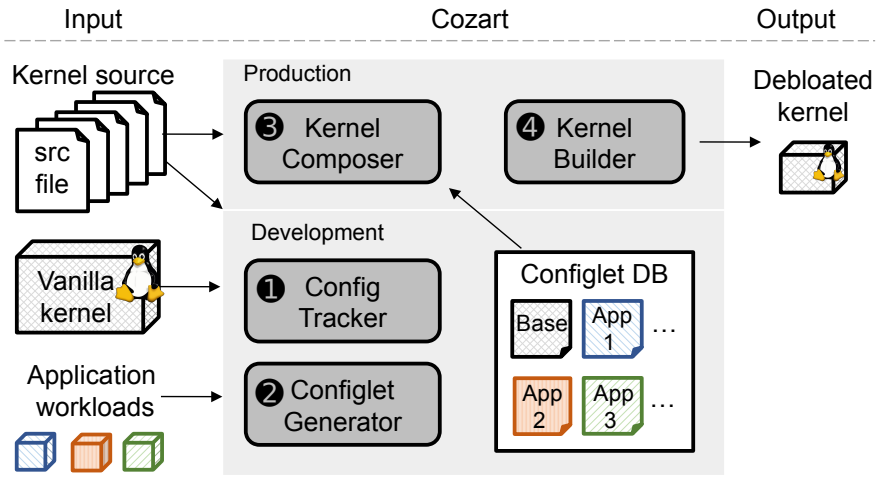


Figure 5.4: Cozart overview: Config tracker records configuration options used by the target application. Configlet generator processes these options into CONFIGLETS and stores them in CONFIGLET DB. Kernel composer takes CONFIGLETS and produces the final configuration. Kernel builder builds the debloated kernel based on the final configuration.

generate the APPLET for the application (which will be stored in CONFIGLET DB). Cozart can later compose multiple APPLETS of different applications during the production phase.

Configuration tracking is done as follows:

1. Tracing program counters (PCs) of target application(s);
2. Mapping PCs to source code;
3. Correlating source code to configuration options.

5.5.2.1 **Tracing Program Counters.**

Cozart uses the Program Counter (PC) register to capture the addresses of instructions that are being executed. My implementation uses `exec_tb_block` provided by QEMU. To ensure that the traced PCs come from the target applications instead of other processes (e.g., background services), Cozart uses a *customized* `init` script that does not start any other applications. The `init` script mounts filesystems (`/tmp`, `/proc` and `/sys`), enables network interfaces (`lo` and `eth0`) and, finally, starts the application directly after kernel boot.

Cozart disables Kernel Address Space Layout Randomization (KASLR) to be able to map the addresses to source code correctly. Please note that the disabling of KASLR is only limited to the configuration tracking phase—*the final debloated kernel can still use KASLR*. In other words, this is not a limitation of Cozart.

5.5.2.2 Mapping PCs to Source Code Statements.

Cozart uses `addr2line` [105] to map a PC to the corresponding statement in the source code to understand which kernel module is used during the kernel execution. `addr2line` utilizes the debugging information compiled in the kernel binary with the baseline configuration.

Loadable kernel modules (LKMs) need additional handling. An LKM is a standalone binary that can be loaded into a kernel during runtime. Since LKMs are not part of the kernel binary, they are not loaded into memory at a fixed address; therefore, `addr2line` cannot be directly applied. In Cozart, I use `/proc/modules` to obtain the start address of each one of the loaded LKMs and then map the PCs to the statement in the LKM binary.

An alternative is to leverage `localmodconfig`, a kernel utility that outputs current loaded LKMs. However, `localmodconfig` only provides information at the granularity of an entire LKM and thus cannot help debloating an LKM based on fine-grained intra-LKM configuration options.

5.5.2.3 Attributing Statements to Configuration Options.

Cozart attributes source code statements to configuration options based on the three patterns presented in Figure 5.1. I implement text-based analysis based on the semantics of C preprocessors and Makefiles. For the C preprocessor based patterns (Figures 5.1a and 5.1b), Cozart analyzes C source files to extract the preprocessor directives (*e.g.*, `#ifdef`) and then checks whether statements within these directives are executed. For Makefile based patterns (Figure 5.1c), Cozart determines if a configuration option should be selected at the granularity of the object files. For instance, in Figure 5.1c, `CONFIG_CACHEFILES` needs to be selected if any of the corresponding files (`cachefiles.o`, `bind.o` and `daemon.o`) are used.

5.5.3 Generating Configlets

Baselets and Applets are generated during the phase in which configuration options are tracked. Cozart marks the end of the boot phase using a *landmark program*—a C program that `mmap`s an empty stub function to a pre-defined "magic address" (I use 0x333333333000 and 0x222222222000 for the start and end) and invokes the stub function. The `init` script described in Section 5.5.2.1 calls the landmark program before running the target application. Therefore, Cozart can recognize the (end of) boot phase based on the magic address in the PC trace.

Cozart takes the configuration options from the application and filters out the hardware-related options that are observed during the boot phase. These hardware features are defined based on their location in the kernel source code (for instance, options located in `/arch`, `/drivers` and `/sound`). I do not exclude the possibility that a hardware-related option may only be observed during the execution of an application, e.g., it loads a new device driver on demand (in such cases, the application is hardware-specific).

5.5.4 Composing Configlets

Cozart composes the Baselet with one or multiple Applets to produce a final kernel configuration that is used to build the kernel. Cozart first unions the configuration options in all the Configlets into an initial configuration and then resolves the dependencies between the configuration options (Section 5.3.2) using an SAT solver. Cozart does not use KConfig, which would silently deselect unsatisfied dependencies during compilation to ensure a valid configuration. To ensure all the selected configuration options to be included in the final debloated kernel, I model the configuration dependencies as a boolean satisfiability problem and use an SAT solver to generate a final configuration—a valid configuration is one that satisfies all the specified dependencies between configuration options. I follow the SAT-based modeling for kernel configuration options described in [96, 106] and use PicoSAT in the implementation of Cozart [107]. It is possible that the SAT solver returns multiple valid solutions. In this case, my current implementation will use the first solution returned from the solver. Note that the first solution may not strictly be the smallest one among all the returned solutions in terms of size; on the other hand, it is straightforward to compile the kernels for all the returned solutions and select the one for the minimal size.

5.5.5 Kernel Builder

Cozart uses the standard kernel build system (KBuild for Linux) to build the debloated kernel based on the kernel configuration composed from Configlets. As kernel building is on the critical path of kernel debloating, Cozart optimizes the build time by leveraging the incremental build support of modern build tool (e.g., `make`). Cozart caches the previous build results (e.g., the object files and loadable kernel modules) to avoid redundant compilation and linking. Upon a configuration change, only the modules with the changes of internal configuration options need to be rebuilt, while the other files can be reused. I find this feature to be particularly useful when a debloated kernel needs to support an additional new application because most applications share many modules but only differ in a small number

Application	Benchmark (Metrics)	Test Suite
HTTPD (v2.4)	Apache benchmark `ab` (RPS)	Apache-Test [109]
NGINX (v1.15)	Apache benchmark `ab` (RPS)	NGINX-test [110]
Memcached (v1.5)	Memtier benchmark (RPS)	Memcached/t [111]
MySQL (v5.7)	sysbench (throughput)	MySQL test [112]
PHP (v7.2)	PHP perf. benchmark (execution time)	PHP-test [113]
Redis (v4.0)	Redis benchmark (throughput)	Redis-test [114]

Table 5.2: Applications and their benchmarks and test suites used in my study All the test suites are the official ones shipped with the open-source projects.

of modules.

5.6 STUDY METHODOLOGY

Target Applications. I use six popular open-source server applications, presented in Table 5.2, that are commonly deployed in virtualized cloud environments. For each application, I use both its official test suite and performance benchmarks to drive the kernel, based on which I use Cozart to generate the Configlets.

Performance Benchmarking Setups. I use the official performance benchmarks listed in Table 5.2 to measure the application performance running in the auto-debloated kernel, in order to verify that the debloating does not introduce any performance regression. All the performance experiments (including the measurement of build times) were performed on a KVM-based VM with 4 VCPUs and 8 GB RAM running on an Intel®Xeon®Silver 4110 CPU operating at 2.10 GHz. I report the benchmark results as the average results of 20 runs.

Baseline Kernel. My baseline kernel is Linux kernel version 4.18.0 from the latest Ubuntu 18.10 Cloud Image (the most widely used Linux distribution, Section 5.3). I also repeated all my experiments on Linux kernel version 5.1.9 from Fedora 30 (the top two most popular Linux distributions deployed in cloud environments [108]). I observe very similar results. Therefore, I only present the results from the Ubuntu 18.10 Cloud Image as the baseline.

Besides the Linux kernel, I also applied Cozart to the L4/Fiasco microkernel and Amazon Firecracker's kernel, as discussed in Section 5.9.

5.7 DEBLOATING EFFECTIVENESS

I use Cozart to debloat the baseline kernel for the target applications listed in Table 5.2. Cozart achieves more than 80% kernel reduction (Table 5.4), which I will discuss in detail in Section 5.8.2. All the debloated kernel can boot and run application workloads, including the corresponding performance benchmarks and test suites.

To validate that the application workloads running on the debloated kernels have normal runtime behavior, I compare the test results and the log messages generated by the applications running on the debloated kernels and the baseline kernel. I verify that the application behaviors are identical. Additionally, I compared statement-level code coverage results of the applications running on the debloated kernel versus the baseline kernel. I find the coverage results are similar with difference within 1% due to certain non-determinism of test cases. In summary, the debloated kernel does not cause any runtime failure of the target applications.

In the remaining of this section, I report the performance of the debloated kernel generated by Cozart with regards to the baseline kernel in the following aspects: (1) boot time reduction, (2) performance benchmark results, and (3) memory savings.

Boot Time. Boot time is critical to provide elastic runtime environments for the serverless computing. The second column in Table 5.3 shows the boot time (and the corresponding reduction with regards to the baseline kernel) for each debloated kernel for the target application. I measure the boot time by reading the value from `/proc/uptime` that contains the duration of the system being on since its last restart. The numbers in the table present the average boot time for ten runs. The kernels debloated by Cozart achieve a time reduction of close to 14%. The reduction is attributed to the savings in load time for a smaller sized kernel image and the skipping of module initialization and device registration for the parts that have been removed (e.g., Fuse file system and Hot Plug PCI controller—both of which may have been removed as part of the debloating).

The boot time depends on the Baselet that, in turn, depends on the original kernel as discussed in Section 5.8.3. A small Baselet will lead to shorter boot times. Surprisingly, Cozart is able to achieve a similar percentage of reduction for Firecracker's kernel with the boot times at less than 100 microseconds.

Application Performance. I benchmark the performance of the applications running on debloated kernels generated by Cozart and compare the benchmark results with the baseline kernel. I do not expect any performance improvements; instead, I look for performance regressions due to removal of performance optimization modules from the original kernels.

Application	Boot Time (ms)	App. Perf. (%)	Memory Save (KB)
Baseline	646	0.00	0
Apache HTTPD	561 (-13.16%)	+1.70%	+4,012
Memcached	557 (-13.78%)	+3.44%	+4,012
MySQL	558 (-13.62%)	+0.29%	+4,016
Nginx	562 (-13.00%)	+3.53%	+4,016
PHP	556 (-13.93%)	+0.21%	+4,012
Redis	556 (-13.93%)	+3.49/3.78%	+4,012

Table 5.3: Boot time, application performance and memory reduction for debloated kernels. The baseline is the original kernel of the Ubuntu 18.10 Cloud Image. The benchmarks for measuring application performance are described in Table 5.2. Note that I show both `get` and `set` requests for Redis.

Note that this is not supposed to happen by the design of Cozart, but I run the performance benchmarks as sanity checks.

Table 5.3 (the third column) shows the improvements of application performance running on debloated kernels with regards to the baseline. There is no performance regression for any of the applications; instead, the debloated kernels show marginal performance improvements for all the applications, mainly because applications load faster and have smaller memory overheads.

Memory Savings. Smaller memory usages can improve resource utilization. Debloated kernels can have the benefit of memory savings. I define memory savings as the reduction in the memory region that hosts the loaded kernel binary, measured by `MemTotal`, as read from `/proc/meminfo`. The numbers shown in Table 5.3 are aligned to page sizes (4 KB in my system). I observe about 4 MB memory savings across the debloated kernels for single applications because the kernel size reduction contributes to memory savings, and all applications have similar reduction percentages (which I will discuss in Section 5.8.1).

To Summarize: The debloated kernels produced by Cozart are all stable—they can directly boot and support the expected application workloads. The debloated kernels can boot 13+% faster than the baseline kernel. The debloated kernel has slightly higher performance compared with the baseline kernel. The debloated kernels achieve more than 4MB memory savings at runtime.

Application	# Kernel Modules		Kernel Size Reduction			
	Static	LKM	Static	LKM	Default	Overall
Ubuntu Vanilla (Baseline)	2443	5001	0%	0%	0%	0%
Base configlet	1212	7	-17.21%	-86.80%	-29.52%	-83.53%
Apache HTTPD	1213	7	-17.19%	-86.80%	-29.50%	-83.52%
Memcached	1215	7	-17.19%	-86.80%	-29.50%	-83.52%
MySQL	1221	7	-17.13%	-86.80%	-29.46%	-83.51%
NGINX	1215	7	-17.19%	-86.80%	-29.50%	-83.52%
PHP	1216	7	-17.21%	-86.80%	-29.50%	-83.53%
Redis	1217	7	-17.17%	-86.80%	-29.49%	-83.52%
Docker	1232	35	-17.02%	-75.11%	-27.30%	-83.01%
Docker + Apache HTTPD	1233	35	-16.99%	-75.10%	-27.27%	-83.00%
Docker + Memcached	1233	35	-16.99%	-75.10%	-27.27%	-83.00%
Docker + MySQL	1239	35	-16.93%	-75.10%	-27.22%	-82.99%
Docker + NGINX	1233	35	-16.99%	-75.10%	-27.27%	-83.00%
Docker + PHP	1236	35	-16.98%	-75.11%	-27.30%	-83.01%
Docker + Redis	1237	35	-16.98%	-75.10%	-27.26%	-83.00%

Table 5.4: Characteristics of the debloated kernel generated by Cozart based on CONFIGLET composition. "Static" and "LKM" refer to all the statically-linked modules and loadable kernel modules, respectively. "Default" refers to a very conservative baseline that only includes the LKMs that are auto-loaded based on Ubuntu 18.10 configuration (Ubuntu includes all the LKMs but does not auto-load every single one by default). "Overall" refers to the overall kernel space (all the modules).

5.8 FINDINGS AND IMPLICATIONS

5.8.1 Boot Phase Visibility

This section discusses the visibility limitation of the state-of-the-art kernel debloating techniques. It also explains how Cozart addresses this accidental limitation by design.

Kernel debloating has to take care of the kernel boot phase in which the necessary kernel modules are loaded to make sure the kernel can start up and run on the deployment environment; otherwise the kernel could either fail to boot or have performance and security regressions due to missing important options that are loaded during the boot phase.

5.8.1.1 **My Experiences with State-of-the-Art Kernel Debloating Tools.**

The state-of-the-art tools, as exemplified by Undertaker [3], do not have an automatic solution to tackle the boot phase. This is because existing tools use `ftrace` for kernel tracing, which can only be started after kernel boot. Therefore, existing tools do not have the *visibility* to the kernel boot phase.

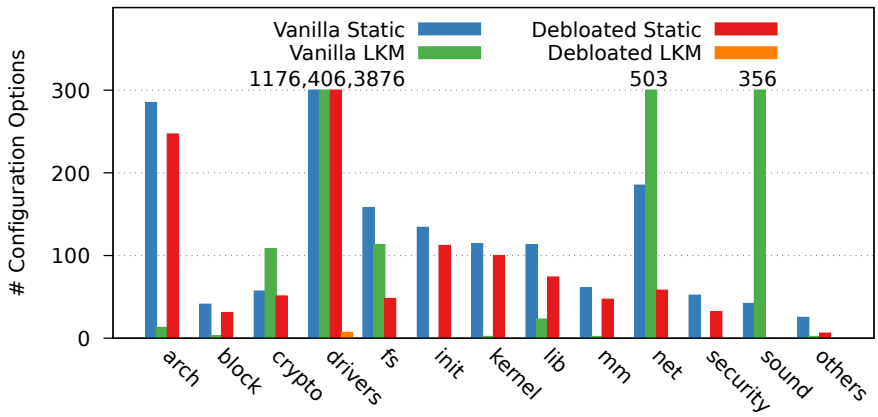


Figure 5.5: Boot-phase configuration options in BASELET and in the baseline kernel. I break down the options based on the Linux kernel source tree.

To work around this limitation, Undertaker turns on `ftrace` as early as possible by extending the system-provided initial RAM disk (`initrd`) [3]. However, I show that turning on `ftrace` at the initialization of RAM disk is not early enough. I show that disk drivers are not captured in Undertaker, and even the RAM disk support itself has to be whitelisted.

To make the debloated kernel bootable, Undertaker eventually requires users to set a whitelist to ensure certain configuration options are included. However, as articulated in [91], "*that brings the user back to the original configuration issue.*" Given the dynamics of kernel configuration (Figure 5.2), it is hard for the handcrafted whitelist to keep updated with the kernel evolution.

I use Undertaker to debloat the same Linux kernel presented in the Undertaker paper [3]. Undertaker produces an unbootable kernel. In addition, serial console options (*e.g.*, `CONFIG_SERIAL_8250` and `CONFIG_SERIAL_8250_CONSOLE`) are missing, which makes the kernel print no messages. I manually add back the serial console, and I find that EXT4 filesystem is misconfigured as an LKM (caused by its design bugs). Besides, the SCSI disk support (`CONFIG_BLK_DEV_SD`) is absent as the current whitelist only preserves an older disk interface, *i.e.*, ATA (`CONFIG_SATA_AHCI`).[3] My experience is consistent with the other report [91]. It shows that using a handcrafted whitelist is not a sustainable solution.

LKTF [51] attempts to address the invisibility to the boot phase using a search-based approach. LKTF fills in configuration options into the unbootable kernel generated by Undertaker until the kernel can eventually boot. It is reported that LKTF takes about five hours to generate a bootable kernel [91]. Although the debloated kernel generated by LKTF has a higher chance to boot (it is reported that the kernel boots in 1/5 of the time [91]),

[3]Despite that the kernel debloated by Undertaker does not boot, I still report its overall kernel reduction of 91.38% as a reference.

LKTF involves building and testing a lot of kernels in VMs until one can successfully boot. I are not able to run LKTF because its implementation is hardcoded to a small number of old kernel versions. The kernel generated by LKTF may have reduced performance and security, as LKTF does not guarantee to include all the original security and performance configuration in the debloated kernel.

5.8.1.2 **Boot-phase Visibility with Cozart.**

Cozart addresses the visibility limitation using a lower level tracing provided by the hypervisor, as discussed in Section 5.5.2. The tracing allows Cozart to observe the complete lifecycle of a kernel and thus include all the modules that are loaded during the boot phase. This brings a key advantage of Cozart: *Every debloated kernel by Cozart can boot and run stably.*

Cozart generates the Baselet for Ubuntu 18.10, which contains 1212 (out of 2443) statically-linked modules and 7 (out of 5001) LKMs. I reduce 1231 static-linked modules and 4994 LKMs compared with the baseline kernel (Ubuntu 18.10). Figure 5.5 breaks it down based on the subdirectories in the kernel source tree. The majority of reduction is from `drivers`, `net`, `sound`, and `fs` (file systems), with a reduction of 94+% in total.

Consider drivers as an example. The vanilla kernel aims to support a variety of hardware and thus contains drivers for different devices (*e.g.*, network cards, PCI and media devices). Cozart only includes the one observed in the traces (*i.e.*, those used by the applications), including `acpi`, `scsi`, `cpufreq`, `tty`, `char`, and `dma` drivers. The vanilla kernel contains different filesystems (`fs`) such as FAT and Fuse. Cozart only keeps EXT4 as the filesystem that results in only 48 `fs` modules in the debloated kernel (e.g., `CONFIG_EXT4_FS`, `CONFIG_EXT4_POSIX_ACL`, `CONFIG_EXT4_ENCRYPTION`) out of 271 in the baseline. The reduced kernel also includes pseudo-filesystems (*e.g.*, `proc` and `sys`) and other `fs` options (*e.g.*, `CONFIG_FS_IOMAP`, `CONFIG_FS_MBCACHE`) to satisfy dependencies.

To Summarize: Existing kernel debloating techniques initialize in-kernel tracing methods (*e.g.*, ftrace) too late and cannot observe the boot phase, which is critical to producing a bootable kernel. Cozart leverages the tracing feature provided by the hypervisor to support end-to-end observability (including the complete boot phase and application workloads) and produce stable kernels that can always boot and run applications stably. The debloated kernel by Cozart does not have any performance regression, as shown in Section 5.7.

5.8.2 Composability

I show that CONFIGLETS can be automatically composed to generate a debloated kernel configuration. A kernel configuration can be composed with one BASELET plus one or more APPLETS. As described in Section 5.5.4, Cozart first unions the configuration options in the CONFIGLETS and resolves configuration dependencies.

5.8.2.1 **Composition Experiments.**

I evaluate Cozart's composability with the following four types of compositions:

- **Individual application.** I use Cozart to compose each individual APPLET and the BASELET to generate a debloated kernel tailored for an individual application.
- **Individual application in a container.** I use Cozart to compose the APPLET along with the CONFIGLET of the Docker Runtime (`dockerd`) in conjunction with the BASELETS.
- **Multi-application.** I use Cozart to compose a debloated kernel to support the LAMP [115] stack (that includes Apache HTTPD, MySQL and PHP). I use a separate application, a MySQL administration tool with a PHP interface, to validate the LAMP stack works, in addition to each individual application in the stack.

5.8.2.2 **Applet Characteristics.**

Cozart successfully generates the APPLETS for all of the target applications listed in Table 5.2. Figure 5.6 shows the breakdown of the APPLETS based on the Linux kernel source tree.

All applications have similar distributions across the subdirectories. The average CONFIGLET size is 117, which is ≈1% of the total number of Linux kernel configuration options. The small percentage is expected because (1) most of the options are for hardware features (81+% are hardware options), and (2) applications typically use a small number of system calls and thus have a small kernel footprint. The results also indicate the benefit of decoupling APPLETS and BASELETS—one can rapidly reconfigure the kernel to accommodate new applications as long as I have the corresponding APPLETS, as shown in Section 5.8.3.

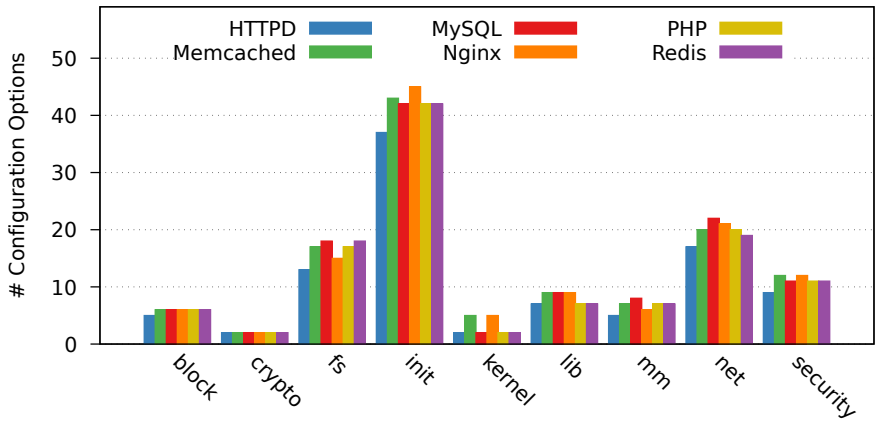


Figure 5.6: Characteristics of APPLETS generated by Cozart for each target application.

5.8.2.3 **Composition Effectiveness.**

Table 5.4 shows the detailed debloating results of the first and second types of composition (single application and application in a container).[4] The reductions are with regard to the baseline kernel. The reason why the reduction numbers look very similar is that 96–99% of the configuration options are from the BASELET generated from the same VM. Cozart can compose the BASELET with the target APPLET to achieve 17+% size reduction for static-linked kernel binaries and 86+% size reduction for loadable kernel modules (LKMs).

I next use Cozart to generate the CONFIGLET for *Docker Runtime* (`dockerd`) using the standard `hello-world` example that starts `dockerd`, pulls the `hello-world` image, and launches a container. I generate debloated kernels by composing multiple APPLETS (including the target application and `dockerd`) and the BASELETS, in a manner similar to how I generate single-application kernels. The debloated kernel can execute all the performance benchmarks described in Table 5.2. The lower half of Table 5.4 shows the reduction achieved by debloated Linux kernels running single applications in Docker containers. Specifically, I can see that the `dockerd` CONFIGLET contains most of the APPLETS already. The debloated kernels have similar reduction ratios.

To compose the kernel configuration for the LAMP stack, I use Cozart to compose the CONFIGLETS of each application (HTTPD, MySQL, and PHP), together with the BASELET to generate a debloated kernel for the entire stack. I deploy and run a separate application, `phpMyAdmin` (a MySQL administration tool with a PHP-based interface). I manually exercise the application to ensure the functionalities of all components (*i.e.*, PHP, MySQL and

[4]I do not focus on security analysis in this paper and, thus, do not report security metrics such as GenSec and IsoSec [3] or vulnerability reduction [17] (those analysis cannot be automated). I limit ourselves to the understanding that a reduction in total code results in an improved security posture due to reduced attack surfaces.

HTTPD) are used. I find no error messages to verify that (1) the kernel supports LAMP stack applications and (2) the kernel supports workloads are different from those used to generate each individual Applet. I find that `phpMyAdmin` works properly without any dysfunctions. The debloated kernel shows a size reduction of 17.13% for a statically-linked kernel binary and 86.80% for LKMs.

To Summarize: The composability of Cozart enables users to generate and maintain application-specific kernel configuration in Applets and deployment-specific kernel configuration in Baselets. A complete kernel configuration can be generated by composing Applets and Baselet.

5.8.3 Fast Application Deployment

A limitation of existing kernel debloating techniques, when one wants to deploy a new application onto a debloated kernel (customized for a different application), is that she has to repeat the entire debloating process from tracing to the re-compilation.[5] Given that debloating a kernel could take hours or days, existing debloating techniques cannot support fast application deployment.

I argue that the aforementioned limitation is caused by the design of existing techniques which make tracing an integral component of the debloating process—the debloating tool has to first trace the kernel by running application workloads and then compile the kernel based on the tracing results. Table 5.5 compares the tracing time and compilation time. I can see that tracing requires significantly more time than compiling. Specifically, the tracing time depends on the application workloads (test suites or benchmarks)—a more complete workloads would lead to higher tracing overhead. For example, MySQL has 5567 test files and requires 86 hours in total to finish the tracing.

With Cozart, tracing can be decoupled from the compilation and thus can be done during the development phase, as long as the results, Configlets, are maintained. *If I have the* Configlets *of the target applications, debloating a kernel takes less than two minutes if compiled from scratch, as shown in Table 5.5.*

[5]One can prepare debloated kernels for every combination of applications and thus save repeated tracing effort. However, this requires prohibitive storage costs due to the combinatorial space. For example, for 10 applications, the potential combinations of applications running on a kernel are 2^{10}.

Application	Tracing Time (second)	Build Time (second)	
		From Scratch	From Baseline
Baseline	0	1035.84	N/A
HTTPD	658.09	119.75	125.47
Memcached	2463.75	119.33	123.92
NGINX	3975.25	119.17	122.52
MySQL	310550.32	119.49	123.37
PHP	14989.02	119.16	123.81
Redis	13004.33	119.57	123.03

Table 5.5: The debloating time for tracing and compilation (from scratch and from a previous baseline build).

5.8.3.1 Build from Scratch.

As shown in Table 5.5, building a debloated kernel from scratch takes less than two minutes. The build is more than 8.5× faster than building the baseline kernel, as the debloating skips more than 83% of kernel code (Table 5.4). The compilation time is proportional to the code size.

I also explore the benefits of *incremental builds* that reusing intermediate files (e.g., object files and LKMs) of a build for the baseline kernel. My hypothesis is that the build for the debloated kernel should be able to reuse many of the object files and speed up. Surprisingly, I find that such practice does not help speed up the build, but instead slowing down the build slightly, as shown in Table 5.5. The reason is that the configuration of the debloated kernel is dramatically different from the baseline kernel, and only a few files can be reused—I observe that more than 99% of files need to be re-compiled. On the other hand, when building from a previous build, KBuild needs to check the reusability of each object file, with the overhead being more than the benefit.

5.8.3.2 Incremental Builds from Debloated Kernels.

I then explore the benefits of incremental builds from debloated kernels, instead of the baseline kernel. I design a few experiments where I want to add a *new* application on top of an existing debloated kernel and measure the additional build time, as shown in Table 5.6.

I find that the incremental builds from debloated kernels are very effective. This is mainly attributed to the fact that many applications share common configuration options and only differ in a small number of options. As a result, Cozart only needs to (re-)compile these small number of additional modules, while reusing the majority of the originally built binaries. As shown in Table 5.6, my framework only needs tens of seconds in the presence of cold

Applications		# New Modules (Options)	Time (Sec.) (Hot/Cold)
Existing	**New**		
Apache	PHP	0	0/0
Apache + PHP	Redis	1	5.67/30.73
Apache + PHP	MySQL	2	18.07/64.78
Docker + Apache + PHP	MySQL	0	0/0

Table 5.6: The number of new options when comparing the new Applet and the composed kernel, and the time it takes to build. Hot/cold refers to hot/cold disk cache.

caches to rebuild the kernel for supporting a new application, while only a few seconds if the cache is hot (typically when dedicated build servers are in use). The incremental build time is determined by the number and size of the new modules that need to be included. Especially, in two cases, the build is entirely skipped, because the Applet of the new application is included in the current Configlets. Note that this cannot be supported without the exposure and analysis of Configlets—in existing debloating techniques, the same cases have to go through the tracing and building process.

To Summarize: Kernel debloating can be done within tens of seconds if the configuration options of target applications are known. The composability of Cozart allows users to prepare Applets during the development phase. To deploy a new application on a debloated kernel, Cozart composes the Applet of the target application with the Configlets of the current kernel to generate a new debloated kernel within tens of seconds.

5.8.4 Coverage

One of the essential concerns of applying existing kernel debloating techniques is the completeness of the debloated kernel regarding the kernel footprint of the target applications. If the target application reaches kernel code that is not included in the debloated kernel, the kernel could potentially crash or returns errors to the applications.

The prior studies on kernel debloating techniques implicitly assume that running a benchmark could cover the kernel footprint of the target applications. Taking web servers as the example, prior studies use simple benchmarks that access static documents to generate the debloated kernel [3].

Table 5.7 shows the statement coverage of each target application when running the official performance benchmark suites. I use `gcov`, an open-source source code coverage analyzer. I can see that the statement coverage reached by the official benchmark is very low—the statement coverage ranges from 6% to 26%. Typically, the benchmarks only exercise the steady states of the target applications, while missing the other part of the application

program (e.g., other features, error handling and recovery states). Take web servers such as HTTPD and NGINX as examples. The benchmarks only exercise the code path for serving static web pages and only lead to a statement coverage of 13% and 8% respectively. As a result, the debloated kernel generated based on those benchmarks can hardly support workloads different from the benchmarks.

Test suites can potentially complement benchmarks in exercising the application's kernel footprint, as test suites are designed to exercise different parts of the software and its use cases. Therefore, I explore the feasibility and effectiveness of using test suites for kernel debloating, given that mature software systems usually provide high-coverage test suites, as shown in Table 5.2.

As shown in Table 5.7, test suites provide much higher code coverage compared with the benchmarks. However, I can see that the code coverage of the test suites is not perfect. For the open-source applications evaluated in this paper, the statement coverage varies from 29%–73%. It is reported, however, that test suites for industrial software have much higher code coverage. For example, Google reports that the code coverage of their software projects are above 80% [116, 117]. I conclude that lack of high-coverage tests is an essential limitation for the adoption of existing kernel debloating techniques based on dynamic tracing. I believe that automated testing techniques such as fuzzing testing [118, 119] and concolic testing [120, 121] and other symbolic execution based test generation techniques [122, 123] can effectively improve the test coverage and alleviate the limitations. On the other hand, those test auto-generation techniques are not silver bullets, *e.g.*, fuzz testing provides few guarantees on coverage, while symbolic execution is challenging to scale.

I compare the Applets generated from benchmarks versus test suites. Table 5.7 shows the size of the two Applets for each application. In general, test suites have higher code coverage and lead to the larger size of Applets, compared with benchmarks. The only exception is HTTPD. The Applet generated from the benchmark has 23 more options,

Application	**Test Suite**		**Benchmark**	
	Coverage	**# Options**	**Coverage**	**# Options**
HTTPD	29%	76	13%	97
Memcached	73%	120	26%	80
NGINX	69%	123	8%	80
MySQL	68%	120	13%	93
PHP	61%	115	6%	35
Redis	57%	115	11%	65

Table 5.7: Statement coverage and number of kernel configuration options of the target applications reached by the official performance benchmarks versus official test suites.

despite lower code coverage. I attribute this to HTTPD's poor test suite.

Interestingly, I find that the configuration options discovered by benchmarks are not always included in the Applets from test suites even for high-coverage test suites such as Memcached. In Memcached, there are three options that only exist in Applets generated by the benchmark. One of them is `CONFIG_PROC_SYSCTL`, which is captured when Memcached reads the maximum number of file descriptors to check whether it can allocate more connections in the scenario where the number of requests overloads the Memcached server.

I would like to note that code coverage cannot guarantee the coverage of the target application's kernel reach (the kernel features needed to support the application). A test suite with 100% statement coverage can cover all possible system calls invoked by the target application, but provides no guarantee to cover all possible system call argument values, or all possible kernel states. Hence, the limitations of dynamic tracing is essential (not accidental) to existing kernel debloating techniques.

To Summarize: An essential limitation of using dynamic-tracing based techniques for kernel debloating (which are the *de facto* approach) is the imperfect test suites and benchmarks. Specifically, the official test suites of many open-source applications have low code coverage. Combining different workloads (e.g., using both test suites and performance benchmarks) to drive the application could alleviate the limitation to a certain extent. On the other hand, code coverage cannot guarantee the coverage of the target application's kernel reach.

5.8.5 Using Domain-specific Information

One essential limitation of existing kernel debloating techniques is the inability to distinguish between kernel modules which are *executed* during the tracing versus those that are *needed*. Current techniques (including Cozart) keep all the kernel modules/code that are used during the boot phase or during application workloads in the debloated kernel. However, modules/code that are executed are not necessarily needed—it is possible that they are reached only because they happen to exist on the execution paths. For example, for virtual machines (VMs), kernel modules/code for BIOS, thermal control, power management are executed but are not needed to boot the VM.

In this section, I study the opportunities of further debloat the kernel based on domain knowledge to remove the kernel modules/code that are not needed for the deployment. I study KVM-based VMs (such as Amazon Firecracker) and Xen VMs. As described in Section 5.3.3, the Linux source code provides configuration templates, `kvmconfig` and `xenconfig`, which provide domain-specific information for KVM and Xen respectively. For

	Static Size	# Remaining Modules	
		Static	LKM
Original debloated kernel	20579167	1215	7
KVM	12302255 (-40.22%)	680 (-44.03%)	1 (-85.71%)
XEN Guest (DomU)	12524616 (-39.14%)	709 (-41.65%)	2 (-71.43%)

Table 5.8: Boot-related kernel reduction on top of the BASELET using domain-specific information (the `kvmconfig` and `xenconfig` templates provided by the Linux kernel).

example, `kvmconfig` uses `virtio` as the I/O interface for both net and block devices, while `xenconfig` uses the `Xen` front-end (`Dom0`) as the main I/O interface; the original BASELET also uses other I/O interfaces such as the SCSI disk driver which are not needed for KVM and Xen VMs.

I use these two templates to replace the original BASELETS to further debloat the already-debloated kernel for KVM (I use Amazon Firecracker as the KVM frontend) and Xen, respectively. Table 5.8 shows the results. *I observe a 40+% and 39+% kernel size reduction of KVM and Xen compared with the original* BASELET*, which is a result of 40+% and 41+% reduction in the number of configuration options for KVM and Xen, respectively.*

The significant reduction comes from the fact that many kernel modules (including partition types, processor features, BIOS, thermal control, power management, ACPI, input devices and legacy serial support) are executed by the baseline kernel at the boot time and thus are included by Cozart. However, those modules are not needed if the deployment is KVM or Xen virtual machines. Therefore, with domain-specific information of these VMs, I can further remove those modules.

To Summarize: Domain-specific information can be used to further debloat the kernel by removing the kernel modules that are executed in the baseline kernel but are not needed by the actual deployment. Take Xen and KVM as examples. I can use Cozart to further reduce the kernel size by 40+% and 39+% based on the `xenconfig` and `kvmconfig` configuration templates provided by the Linux kernel source.

5.9 GENERALITY

To the best of my knowledge, all existing kernel debloating tools and studies only focus on the generic Linux kernel. I report my experience in debloating other types of OS kernels by porting Cozart to the L4/Fiasco microkernel and the Firecracker kernel.

5.9.1 Cozart Portability

Cozart is portable to other types of OS kernels. I have ported Cozart to debloat the L4/Fiasco microkernel and the Firecracker kernel. I apply Cozart on the L4/Fiasco microkernel that also uses KConfig as the configuration language with minor changes to a newer source tree layout and different build steps. I modify the kernel bus from MMIO to PCI for QEMU tracing and change the bus back to boot on Firecracker.

My experience shows that configuration-driven kernel debloating techniques are independent of kernel architectures, regardless of whether they are monolithic kernels or microkernels. The porting only requires the integration with the kernel configuration system (e.g., KConfig) and the kernel build system (e.g., KBuild).

5.9.2 L4 Microkernel

I apply Cozart to the L4/Fiasco microkernel [124]. The result is surprising—the debloated kernel is 47% smaller when compared to the default. The reason for the significant reduction is that the default L4/Fiasco kernel configuration enables a heavy kernel debugger, `CONFIG_JDB`, that contributes to a big part of the final kernel. The debloated kernel no longer houses this module. I would like to note that it may sound straightforward that removing a debugger makes the kernel smaller in size; however, without an automated solution like Cozart, it is impractical to examine and select every single configuration option manually, given the hundreds and ever-increasing number of configuration options (L4/Fiasco has 218 configuration options). This has been repeatedly validated by recent user studies on system configuration management [89, 125, 126, 127].

I choose to run an ambient occlusion renderer benchmark, aobench [128], as none of the applications in Table 5.2 can directly run on L4/Fiasco. I run aobench in a KVM-based VM with 1 VCPU and 128 MB RAM and observe a performance improvement of 4%.

5.9.3 Firecracker

While still based on Linux, Amazon Firecracker's kernel is a special case, as the kernel is extensively minimized to optimize for Function-as-a-Service workloads (Firecracker does not support generic Linux kernel). I apply Cozart on the Firecracker kernel using the target applications listed in Table 5.2 to understand the space for application-specific kernel debloating like Cozart to benefit an extensively optimized kernel like Firecracker.

Overall, Cozart reduces the number of kernel configuration options from 910 to 729 (a 20%

reduction), leading to a 19.76% kernel size reduction and an 11% faster boot time (reduced from 104 microseconds to 92.5 microseconds).[6]

To Summarize: Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and extensively customized kernels (e.g., the Firecracker kernel). The reduction is significant: 47.0% for the L4 microkernel and 19.76% for the Firecracker kernel.

5.10 CONCLUSION

OS kernel debloating techniques have not received wide-ranging adoption due to the instabilities of the debloated kernels, the lack of speed, completeness issues, and the requirements for manual intervention. This paper studies the practicality of OS kernel debloating techniques with the goal of making kernel debloating practical in real-world deployments. I identify the limitations of existing kernel debloating techniques that hinder their practical adoption. To understand these limitations, I build Cozart, an automated kernel debloating framework which enables us to conduct a number of experiments on different types of OS kernels with a wide variety of applications. I share my experience and solutions in addressing the accidental limitations that exist in the existing kernel debloating techniques and discuss the challenges and opportunities in addressing the essential limitations that are critical to the practicality of OS kernel debloating.

I have made Cozart and the experiment data publicly available at: `https://github.com/hckuo/Cozart`

[6] I use Firecracker's own mechanism to measure the boot time instead of reading from `/proc/uptime`. Firecracker uses a pre-defined I/O port to notify boot completions.

CHAPTER 6: BEYOND TRADITIONAL KERNEL DEBLOATING

6.1 OVERVIEW

In this chapter, I expand the definition of debloating. Restricting access to kernel code and reducing inefficiency in the kernel can also be considered as means of debloating. I make the observation that extensions can be collectively optimized after they are verified individually and loaded into the shared kernel. I present KFuse, a framework that optimizes Linux kernel extensions (BPF) and brings significant performance gains.

6.2 INTRODUCTION

Operating system (OS) extensions are more popular than ever. Linux BPF is marketed as a "superpower" that allows user programs to be downloaded into the kernel, verified to be safe and executed at kernel hook points. Currently, BPF extensions are used for system call security (*e.g.*, seccomp-BPF [129]), performance tracing (*e.g.*, tracepoints [130] and bcc [131]), and network packet processing (*e.g.*, express data path (XDP) [132, 133]); many other use cases are proposed recently [131, 132, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154]. BPF extensions have high performance, because BPF code is highly optimized [155] and BPF programs are executed entirely in the kernel. For example, sandboxing system call with seccomp BPF filters is reported to be 2× faster, compared with ptrace-based system call filtering [156]; using XDP [132, 133] to implement firewall rules improves network throughput by 11× compared to iptable-based implementations [157].

Currently, BPF extensions are individually verified by an in-kernel BPF verifier and then executed independently. A common execution pattern for BPF extensions is to execute a *chain* of independently-loaded BPF extensions that are attached at the same hook point. The chain pattern is a natural implementation choice to support BPF extensions that are (1) installed by multiple principals [129, 130, 158]; (2) installed at different points of time (*e.g.*, for temporally-specialized security policies [159, 160]), and (3) maintained to be modular and separate (for maintainability and debuggability) [161].

More fundamentally, due to the limited scalability of BPF program verification, the BPF verifier enforces each BPF extension to be small in size—each BPF program is limited to 4096 instructions with a maximum stack size of 512 bytes; therefore, large BPF features need to be split into small BPF programs, each of which can be verified in time [162].

However, I observe that the chain pattern has large performance overhead, due to indirect jumps penalized by security mitigations (*e.g.*, Spectre), loops, and memory accesses. The typical chain pattern is implemented by a sequence of *indirect jumps*. The kernel maintains an array of pointers of loaded BPF extensions which are called one by one via indirect jumps to these pointers. While indirect jumps were optimized in modern CPUs via branch prediction, recent security mitigations to speculative vulnerabilities [163] (Spectre, specifically Variant 2) incur significant overhead to indirect jumps, negatively affecting the execution of the chained BPF execution. For example, the *de facto* software mitigation, Retpoline [164], makes indirect jumps 13.3× slower [5, 165]. Besides loops, BPF extension chains can also be formed by tail calls; a BPF program can tail-call another BPF program, which takes a memory access and an indirect jump.

To make the matter worse, the overhead of BPF extension chains increases with the increase of the length of the chain. I have seen real-world BPF use cases with long BPF extension chains. For example, `systemd` [166] installs 19 seccomp BPF filters to its services. My benchmark on Redis initialized by `systemd` shows that the chained BPF extensions can cause a 10% slowdown of Redis. I expect such long chain to be commonplace in future BPF use cases for fine-grained policies and richer features. The overhead of the chains would be untenable.

In this paper, I argue that execution and verification should be a separation of concerns. I propose to decouple the execution of BPF extensions from their verification requirements and limits—kernel extension programs can be collectively optimized, after each extension is individually verified and loaded into the *shared* kernel. I demonstrate that the decoupling could lead to new opportunities for performance optimizations, while maintaining the safety of the verified extension programs.

I present KFuse, a framework that dynamically and automatically optimizing chains of BPF extensions by transforming indirect jumps into direct jumps, unrolling loops, and saving memory accesses, without loss of security or flexibility. Specifically, KFuse merges a chain of BPF extensions into a fused BPF program by rewriting the return instructions into direct jump instructions and updating the jump offsets. For loop-based chains, KFuse updates the return values based on the original loop semantics (tail calls do not return). KFuse ensures that the fused BPF program maintains all the safety properties required by the BPF verifier. This can be proved by induction-based on the safety properties of the original BPF extension programs in the chain (ensured by the BPF verifier). Lastly, KFuse is fully transparent to user space. It achieves the transparency by maintaining the original verified BPF program and only performing the optimizations for execution. I integrate KFuse into three Linux kernel subsystems that adopt BPF: seccomp, tracepoint, and cgroup. I show that KFuse

```c
for (; f; f = f->prev) {
    u32 cur_ret =
↪   bpf_prog_run(f->prog,
↪   sd);
    if (cur_ret < ret) {
        ret = cur_ret;
↪   *match = f;
    }
}
```

(a) Seccomp execution loop

```c
u32 _ret = 1;
_item = &_array->items[0];
while ((_prog =
↪   _item->prog)) {
    _ret &= func(_prog,
↪   ctx);
    _item++;
}
return _ret;
```

(b) Tracepoint execution loop

```c
while ((_prog =
↪   READ_ONCE(_item->prog)))
↪   {
    ...
    _ret &= func(_prog,
↪   ctx);
    ...
    _item++;
}
```

(c) Cgroup execution loop

Figure 6.1: **Common patterns of executing BPF extensions:** A chain of BPF programs are store in an array or a list and are executed in loops of indirect jumps.

can be easily integrated in existing BPF use cases—it takes only 20, 28, and 34 lines of code changes for the three subsystems respectively.

I evaluate the effectiveness of KFuse in optimizing execution of BPF extension chains with real-world BPF use cases (the `systemd` BPF filters) as well as synthetic BPF extensions for scalability analysis. KFuse demonstrates 85% performance improvement of BPF chain execution and 7% of application performance improvement over existing BPF use cases (`systemd`'s seccomp BPF filters). When the chain is long (160 BPF filters), KFuse is able to improve NGINX performance by as much as 2.3 times.

This paper makes the following contributions:

- I propose a new perspective of optimizing a chain of verified BPF extensions collectively;
- I benchmark the overhead of executing BPF extension chains, based on which I identify the root causes and analyze their performance impacts;
- I present KFuse, an in-kernel framework for effectively optimizing BPF extension chains.

I will release all the code and dataset in the paper.

6.3 BPF EXTENSION AND THE CHAIN PATTERN

In this section, I introduce BPF extensions for the Linux kernel and identify how admitting extensions into the shared kernel creates a common chain pattern where multiple extensions are attached to the same hook.

6.3.1 BPF extensions

Linux supports kernel extensions implemented in the BPF (Berkeley Packet Filter) language [167]. BPF was originally used for network packet filters; later, it is adopted by many kernel subsystems, including security [129, 168], tracing [130, 131, 169], and process management [158]. Recent research has proposed a variety of BPF use cases, including storage [170], virtualization [171], hardware offloading [132, 172], and others [139, 173]. Note that in Linux, "BPF" stands for extended BPF (or *eBPF* in short). While some subsystems, such as seccomp, still use the classic BPF (cBPF) language, they are converted into the eBPF bytecode at the execution time. So, I focus my discussion on eBPF.

BPF has ten registers and a 512-byte stack. Instructions are 9-byte long and implement a general-purpose RISC instruction set. When an extension is loaded into the kernel, Linux compiles its BPF instructions "just in time" (JIT) into native instructions for better performance. The JIT compilation is optional. BPF programs maintain states via BPF maps. BPF maps are kernel objects that are in the form of key-value pairs, which can also be exposed to user space.

Every BPF program needs to pass an in-kernel BPF verifier; BPF programs that fail the verification will be rejected. The verifier ensures that the BPF extension program does not have unbounded loops, unreachable code, or out-of-bound jumps. So, a verified program is guaranteed to terminate, never jump to invalid locations and never access invalid out-of-bound memory [162]. To make sure the verification can be done in time, a BPF extension is limited to 4096 instructions.

In order to support more complex, modular extension programs, BPF includes a *tail call* mechanism that allows one program to call another one without returning. These programs are verified independently. The stack frame from the old program is unwound and reused. A program array that specifies available programs to tail call needs to be populated first. The program then reads the array and tail calls the target program.

In this paper, I focus on three Linux kernel subsystems that support BPF extensions, seccomp [129], tracepoint [130] and cgroup [158]. Seccomp uses BPF extensions for system call filters that restrict the system calls and their argument values a process can invoke. Tracepoint enables BPF extensions to observe the kernel states for profiling and debugging. Cgroup (control groups) limits the resource usage of processes and filters network traffic with BPF extensions.

6.3.2 Emergence of BPF chain patterns

A chain of BPF extensions is formed when multiple extensions are attached to the same kernel hook point. This is typically implemented in the form of loops. Figure 6.1 shows three simplified examples from seccomp (Figure 6.1a), tracepoint (Figure 6.1b) and cgroup (Figure 6.1c). Another way to form a chain is through tail calls—a BPF program (the caller) can tail-call another BPF program (the callee), and the callee can tail-call other BPF programs.

I categorize four reasons of the chain pattern: (1) extensions are installed from different principals, (2) developers enforce modularity of kernel extensions, (3) extensions are installed at different points in time, and (4) extensions need to be verified independently.

Extensions installed by different principals

Extensions from different principals can not be combined together before being loaded to the kernel, because they may not trust or even be aware of each other. These extensions are verified independently and stored in an array of extensions, and, therefore, a chain is formed. Take seccomp-BPF as an example. Seccomp BPF filters can also be installed from different principals. In a cluster management system, host-specific seccomp BPF filters are installed [174]; the container runtime (*e.g.*, Docker) can install container-wide BPF filters; and applications can install application-specific BPF filters. All these filters are attached at the same location (the system call entry point). In tracepoint, multiple extensions can be attached to the same tracepoint. Processes from different principals collect performance statistics by installing their own tracepoint extensions. Another example is infrared (IR) decoding—decoding protocols are expressed as BPF extensions that can come from different principals.

Modularity of extensions

Modularity is a reason why developers maintain extensions to be single-purpose and separate. A systemd developer writes in the mailing list [161] regarding the usage of seccomp filters: *"Keeping the filters separate made things a lot easier and simpler to debug, and my log output and testing became much less of a black box. I know exactly what worked and what didn't, and my test validates each filter"*. Cgroup firewall rules are also easier to maintain, debug and reuse when these rules are simple. For instance, a common way to build a firewall is to have a rule that denies all packets and rules to allow specific traffic. Each rule is

implemented as a BPF extension, which can be reused and maintained for different policies. One use case of the tail call is to separate big and complicated extensions into smaller and modular ones and chain them together.

Extensions installed at different points in time.

Not all extension programs are known a priori so they can not be combined before being loaded. For example, *temporal system call specialization* [159, 160] is known for security benefits by installing filters at different execution phases of the application (*e.g.*, the initialization phase and the serving phase). Recent work shows that temporal system call specialization can reduce 51% more of the attack surface, in terms of the system call interface. Seccomp ensures that a BPF filter, once installed, cannot be removed during the process lifecycle. Therefore, temporal specialization leads to multiple seccomp filters to be installed at different points in time, forming a BPF program chain. Besides seccomp, tracepoint and cgroup also allow BPF extensions to be dynamically installed at different points in time, on demand.

Extensions that need to be verified individually.

A more fundamental reason of the chain pattern comes from the limited scalability of BPF verification. The verification time depends on the size and complexity of the BPF program. To make sure that a user-supplied BPF extension can be verified in time, the verifier imposes limitations on the size and complexity of the BPF extensions, including the number of instructions, the number of jumps, the stack size, etc. Therefore, to pass the verification, a large, complex BPF extension need to be split into small-sized programs in a chain.

6.4 THE COST OF CHAINING EXTENSIONS

In this section, I benchmark the overhead of executing a BPF extension chain. The main overhead comes from the *indirect jumps* that are invoked to switch extensions on the chain and the *loops* that are used to iterate extensions.

6.4.1 Methodology

I design the benchmark for measuring the overhead of a chain of BPF executions. A chain can be formed when multiple extensions are attached to the same kernel hook point, or when

```
void func1(){}
void func2(){}
void func3(){}
void (*funcs[3])() =
↪  {func1, func2, func3};
start = get_timestamp();
for(int i = 0; i < 3; i++)
↪  funcs[i]();
end = get_timestamp();
```

(a) Indirect jumps in a loop

```
void func1(){}
void func2(){}
void func3(){}
void (*fptr1)() = func1;
...
start = get_timestamp();
(*fptr1)(); (*fptr2)();
↪  (*fptr3)();
end = get_timestamp();
```

(b) Indirect jumps with an unrolled loop

```
void func1(){}
void func2(){}
void func3(){}
start = get_timestamp();
func1();
func2();
func3();
end = get_timestamp();
```

(c) Direct jumps with an unrolled loop

Figure 6.2: **Benchmarks for analyzing the cost of executing a chain of BPF extensions: (a)** measures the cost of loops and indirect jumps. **(b)** measures the cost of indirect jumps. **(c)** measures the cost of direct jumps. Comparing **(a)** and **(b)** to obtain the cost of loops. Comparing **(b)** and **(c)** to obtain the cost of indirect jumps.

a extension tail-calls other extensions. When multiple BPF extensions are attached to the same hook point, these extensions are executed in a loop (exemplified in Figure 6.1). The loop iterates a program array and indirectly jumps to each item on the array. I measured `call` instruction as an approximation of `jump` instruction (Figure 6.2) so the results include jumping to the target address, setting up the stack and pushing the return address on the stack.

I firstly establish the baseline and measure a chain of N extensions by calling indirect jumps to 3 different functions in a loop via function pointers, as shown in Figure 6.2a. Then I unroll the loop, as shown in Figure 6.2b, to obtain the loop cost by comparing the results from Figure 6.2a and Figure 6.2b. Finally, I convert indirect jumps to direct jumps, as shown in Figure 6.2c to obtain the indirect jump cost by comparing the results from Figure 6.2b and Figure 6.2c. A extension that uses tail calls needs to load the target program from a program array. This operation incurs memory access cost which is also part of the cost of executing extensions by a loop. Therefore, I reuse the benchmark and present the result in Section 6.4.2.3

I conduct all the benchmarking experiments on a single server with a 16 core Intel Xeon Silver 4110 CPU at 2.10GHz and 64 GB of memory. I use the `rdtscp` instruction to read the CPU cycles (*i.e.*, `get_timestamp`). I use `gcc`[1] to compile my benchmark programs with optimization level `-O0`.

[1]https://gcc.gnu.org/

	Retpoline	No Retpoline
Loop + Indirect jumps (Fig. 6.2a)	46.3	7.7
Unrolled + Indirect jumps (Fig. 6.2b)	32.6	2.6
Unrolled + Direct jumps (Fig. 6.2c)	2.6	2.6

Table 6.1: **Cycles needed to call a BPF extension:** Indirect jumps incur 30 additional cycles with retpoline. Loops incur 13.7 additional cycles with retpoline and 5.1 without.

6.4.2 Benchmark results

Table 6.1 summarizes my main results. In short, executing a function directly with unrolled loops takes 2.6 cycles; executing a function in loops with indirect jumps takes up to 46.3 cycles when retpoline is enabled, a 17.8× increase.

6.4.2.1 Indirect jump cost.

The common pattern of executing a chain of BPF extensions is to execute *indirect jumps* to locations where these extensions are loaded. An indirect jump (as known as an indirect branch) is a jump instruction with an argument specifying where the target address is located (*e.g.*, `jmp rax` jumps to the location specified by `rax`) instead of the argument being the target address as in direct jumps (*e.g.*, `jmp 0xdeadbeef` jumps to the address `0xdeadbeef`). To measure the cost of indirect jumps compared to direct jumps, I call functions indirectly by function pointers as shown in Figure 6.2b.

Security mitigations for speculation vulnerabilities (*e.g.*, Spectre [163]) significantly increase the cost of indirect jumps and make it expensive to call a chain of BPF extensions via a loop of indirect jumps. This is because indirect jumps are typically optimized by speculation—modern CPU architecture speculates the target address for indirect jumps in the pipeline for optimal performance; however, with Spectre [163] and Meltdown [175] which exploit hardware design flaws of speculation (to leak information from unintended speculative paths via micro-architectural side channels), the speculation is disabled. Fixing these vulnerabilities without performance impact remains as an open problem.

I benchmark the *de facto* software mitigation for speculation vulnerabilities—Retpoline [164], and discuss the other alternative mitigations proposed by recent research as they are not readily available for benchmarking.

Retpoline. Figure 6.3 shows the implementation of retpoline. The key idea of retpoline is to prevent CPUs from speculating the target address of an indirect jump. Retpoline

```
1.    call retpoline_call_target
2. 2:
3.    lfence /* stop speculation */
4.    jmp 2b
5. retpoline_call_target:
6.    lea 8(%rsp), %rsp
7.    ret
```

Figure 6.3: Retpoline assembly

works by placing the target address on top of the stack and calls a *thunk* to jump to target by a return instruction (line 7). The loop in the middle (line 2–4) is never executed, and the purpose of it is to fill the instruction pipeline with dummy values to prevent the CPU from speculating the actual target address, effectively nullifying the substantial performance benefits of speculation along with the risks.

As shown in Table 6.1, without retpoline, the indirect jump with a function pointer takes 2.6 cycles which is as fast as direct calls in Figure 6.2c. The is due to the high accuracy (up to 98% [176]) of branch target predictors in modern CPUs. I also observe that the branch miss rate is merely 0.01%. However, with retpoline enabled, an indirect jump takes 30 extra cycles making it 12.5× slower because retpoline forces a branch prediction miss on every indirect jump and stops the CPU from speculatively executing more instructions (*i.e.*, 100% branch miss rate). Also, retpoline itself introduces more instructions including jumping to the retpoline thunk, moving the target function on the stack and returning to the target function. In particular, jumping to the thunk takes 14.79% of the sampled cycles, moving the target function on the stack takes 8.87% and returning to the target function takes 76.33%.

Alternative mitigations. Recent work [177, 178, 179] proposes to use control-flow integrity (CFI) to regulate speculation targets, and to use indirect call promotion (ICP) as an optimization to avoid indirect jumps. CFI-based approaches need hardware supports to speculate only known targets for protection against Spectre-like attacks. When misspeculation happens, an expensive serializing instruction (*e.g.*, `lfence`) is used.

Promoting indirect calls in the loops I target (*e.g.*, through LLVM [180] or BOLT [181]) would produce different code. ICP needs to predict the indirect branch target and convert an indirect jump into a conditional jump *i.e.*, a comparison and a direct jump. Its performance depends on the prediction. An indirect jump with retpoline is still taken for misprediction. On the other hand, KFuse results in one indirect jump (to the merged program) and unconditional jumps (internally stitching together that program). My narrowed scope (focusing

on extensions rather than more general techniques) enables an approach with unconditional jumps.

6.4.2.2 Loop cost

A loop for executing a BPF extension chain includes the following operations. An index counter represents the index in the current iteration. The counter needs to be incremented by 1 to find the next program at the end of each iteration. Within each iteration, the index counter is used to retrieve the corresponding element in the array. Components of a loop result in more instructions and branches compared to the unrolled loop. To measure the cost of the loop, I manually unroll the loop in Figure 6.2a and call functions via function pointers as shown in Figure 6.2b. I compare the two cases to obtain the extra cycles per iteration that the loop contributes which are 13.7 (29.6% of the extension call) with retpoline and 5.1 (66% of the extension call) cycles without. Therefore, the loop can cause overheads for executing BPF extension chains. Note that retpoline stalls CPU pipelines and magnifies the cost.

I further investigate and break down the cost of a loop. The cost of a loop per iteration includes four parts, (1) checking if the loop condition holds to determine if the loop is over (usually on the index is less than the array size), (2) incrementing the index counter, (3) loading the extension pointer from the array based on the index counter, and (4) jumping to the beginning of the loop. I use `perf` to profile the loop by CPU sampling. Checking the loop condition takes 28.91% of the sampled cycles. 51.76% of the cycles are spent incrementing the index counter stored on the stack. Loading extensions from the array takes 10.99% of the sampled cycles. Finally, 8.34% of the cycles are spent jumping back to the beginning of the loop. The common approach to mitigate the cost is to unroll the loop. However, this obvious solution of loop unrolling can not be done, if the execution of BPF extensions are tightly coupled to the verification and admission of them.

6.4.2.3 Memory access cost

To use tail calls, a program needs to firstly look up the target program from a program array and jump to the target. Looking up the target program is essentially the same operation as one of the loop operations – loading the extension from the array based on the index counter (Section 6.4.2.2). These two operations both read the target program address from an array and therefore, need memory access. According to my measurement of loops, I find that the memory access cost can take up 11% of the loop cost which is 1.5 cycles with

retpoline.

6.4.3 Discussions and Implications

The chain overhead can be significant when the BPF extensions are placed at a performance-critical path. For instance, 54% performance regression is reported for XDP packet filtering when there are only two indirect function calls (jumps) per packet [182]. When the chain is long, the overhead will be accumulated and impact application performance. Some real-world use cases have much longer BPF extension chains than the three-extension chain used in my measurement. For example, `systemd` [166] installs tens of seccomp BPF filters for different system services, which has significant performance overhead (as shown in Section 6.6.1.1).

The chain overheads come from the fact that BPF extensions are individually verified and loaded into the kernel. I argue that the execution of these extensions should not be constrained by how they are verified and loaded. Instead, the execution of the chained BPF programs can be *collectively* optimized in the kernel. Specifically, I can see that replacing indirect jumps with direct jumps, unrolling loops, and saving memory accesses, and each leads to in remarkable performance improvements (Table 6.1). Therefore, in KFuse, I start from supporting these three optimizations.

6.5 KFUSE DESIGN AND IMPLEMENTATION

6.5.1 Overview

KFuse is a in-kernel framework for *collectively* optimizing verified BPF extensions that are installed at the same hooking point, in the pattern of chains. In essence, it separates the concerns of verification and execution of BPF extensions. BPF extensions that are installed by different principals or at different points of time are verified *independently* to ensure safety. KFuse is invoked after the verification to optimize the execution performance.

Currently, KFuse supports three generic optimizations: (1) converting indirect jumps into direct jumps, (2) unrolling loops, and (3) saving memory accesses by rewriting BPF tail calls. It can be easily extended to incorporate other optimizations, such as domain-specific ones.

A BPF extension is loaded from user space, verified by the kernel, and optionally just-in-time (JIT) compiled to native machine code. Then, the target kernel subsystems (*e.g.*, seccomp, tracepoint, and cgroup) can install it by attaching it to the corresponding hook points. KFuse can be invoked dynamically with the input of a chain of BPF extensions; it

generates one optimized BPF extension that "fuses" the original BPF extension chain and replaces it with the fused extension at the hook point. Currently, I invoke KFuse after a new BPF extension is loaded and verified. KFuse fuses BPF programs at the BPF-instruction level, rather than native machine code—the fused BPF extension can still be (optionally) JITed.

Figure 6.4 shows how KFuse is invoked in seccomp. It invokes the currently-attached BPF filter (`current_filter`) with the newly-admitted BPF filter (`prepared`). The current filter can be a fused BPF program.

```
1.  prepared = seccomp_prepare_user_filter(filter);
2.  if (current_filter != NULL) merge_bpf_progs(current_filter->prog, prepared->prog,
3.          SECCOMP_RETURN_POLICY);
```

Figure 6.4: KFuse for BPF seccomp system call filters

Figure 6.5 further illustrates the process, where Figure 6.5a illustrates the original BPF chains at different hook points and Figure 6.5b illustrates the fused BPF extension when KFuse is used. KFuse first fused `b`, `c`, and `d` at the hook point `H2` into an optimized program, `bcd`, and `f` and `g` at `H3` into `fg`. When a new BPF extension `E` is loaded and hooked at `H3`, it fuses `fg` and `E` into `fgE`.

The fused program maintains the safety properties enforced by the BPF verifier, which can be proved by induction-based on the safety properties of the original BPF program (ensured by the verifier). On the other hand, the fused program is not limited by the size or complexity constraints required by the BPF verifier to bound the verification time.

KFuse is an in-kernel mechanism that are transparent to user space. It preserves the original extension structure and creates the illusion of a chain of multiple BPF extensions. BPF extensions are loaded and installed in exactly the same way and no user-space change is needed.

Why not fuse at user space? Fusing the BPF extensions at user space is not an option. The fused BPF extension could be difficult for the existing BPF verifier to verify due to the increased complexity. Furthermore, certain BPF use cases (*e.g.*, seccomp) do not allow dynamic updating installed BPF extensions for security reasons.

6.5.2 Optimizing a loop of BPF extensions

As discussed in Section 6.4, using a loop to iteratively execute loaded BPF extensions is the most common implementation pattern to support BPF extension chains. KFuse optimizes

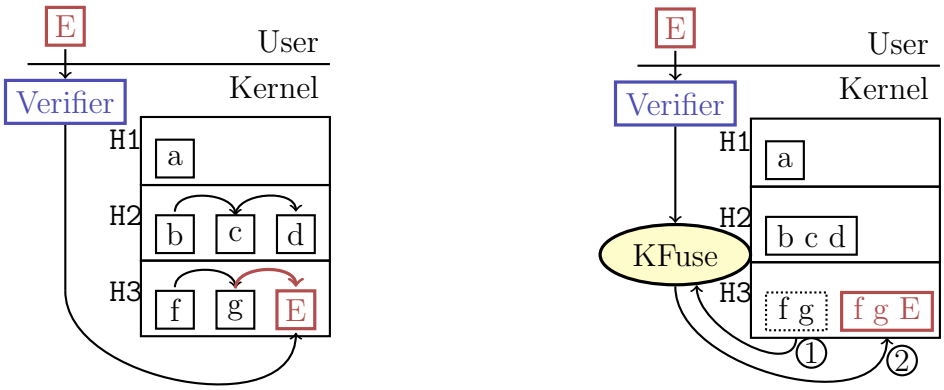


Figure 6.5: Illustration of KFuse: (a) the original BPF extensions; (b) the optimized BPF extensions with KFuse.

such patterns by merging the BPF extension chain into one large BPF extension by unrolling the loop and converting indirect jumps into direct jumps.

Conceptually, such an optimization is as simple as concatenating two BPF extensions. Two important issues need to be taken care of: *return instructions* and *jump offsets*.

6.5.2.1 Rewriting return instructions into direct jumps.

A return instruction terminates the execution of a BPF program. To merge two BPF programs `A` and `B` into `A+B`, KFuse needs to automatically rewrite the return instructions in `A` into two conceptual instructions, one "jump" instruction destined to the first instruction in `B` and one "update" instruction for updating an aggregated return value. The "jump" instruction can be implemented using a *direct* jump. The "update" instruction may need multiple instructions to implement.

KFuse locates every return instruction in the BPF program and replaces it with a direct jump instruction destined to the start of the next BPF program. If the BPF program is the last one in the chain, KFuse replaces the return instructions with direct jumps to a global return instruction (which is the last instruction of the merged program, created by KFuse).

Figure 6.6 shows an example of merging two BPF programs for seccomp, `FILTER_1` and `FILTER_2`. KFuse replaces each return instruction in `FILTER_1` with a direct jump instruction destined to `FILTER2_START`, and replaces each return instruction in `FILTER_2` with a direct jump destined to `END`.

6.5.2.2 Updating return values.

Updating return values needs to follow the loop semantics to ensure the fused program has the same semantics as the original loop. Different applications aggregate the return value of individual BPF programs in the loop differently, as shown in Figure 6.1. I refer to the aggregation semantics as a *return value policy*.

Currently, KFuse supports all the loop semantics of BPF use cases in Linux, including returning the minimal value (seccomp, Figure 6.1a), returning the bitwise-AND of the values (tracepoint and cgroup, Figures 6.1b and 6.1c), and returning the first non-zero values (`tc` and `lsm-bpf`). I discuss the return value policies of seccomp, tracepoint, and cgroup:

- **Seccomp.** A seccomp filter returns an action such as allowing the system call or killing the process. The more restrictive the action is, the smaller the return value is. Consequently, I implemented a return value policy that initializes the return value to the maximum integer value and only updates the new return value only if it is smaller than the current value to match the semantics of seccomp (Figure 6.1a).

- **Tracepoint.** The final return value of tracepoint BPF programs is calculated by performing an AND operation on each return value of the individual programs. The return value policy is implemented by initializing the return value to 1 and updating it via an AND operation on the current and new values.

- **Cgroup.** When a network packet arrives to a process in a cgroup, programs attached to the cgroup will be executed to determine the action for the packet based on the return values of BPF programs. The final value is also calculated by an AND operation like tracepoint. Therefore, the return value policy is the same as the one for tracepoint (initializing the return value to 1 and updating it via an AND operation on the current and new values).

KFuse initializes the global return value stored in temporal storage, and I use `BPF_REG_AX` in my implementation. It updates the return value at the locations of the original return instructions based on the return value policy (which is an input of KFuse). The global return value will be returned by the fused BPF program. In Figure 6.6, the return values are updated based on seccomp's return value policy that returns the smallest value (the most restrictive policy).

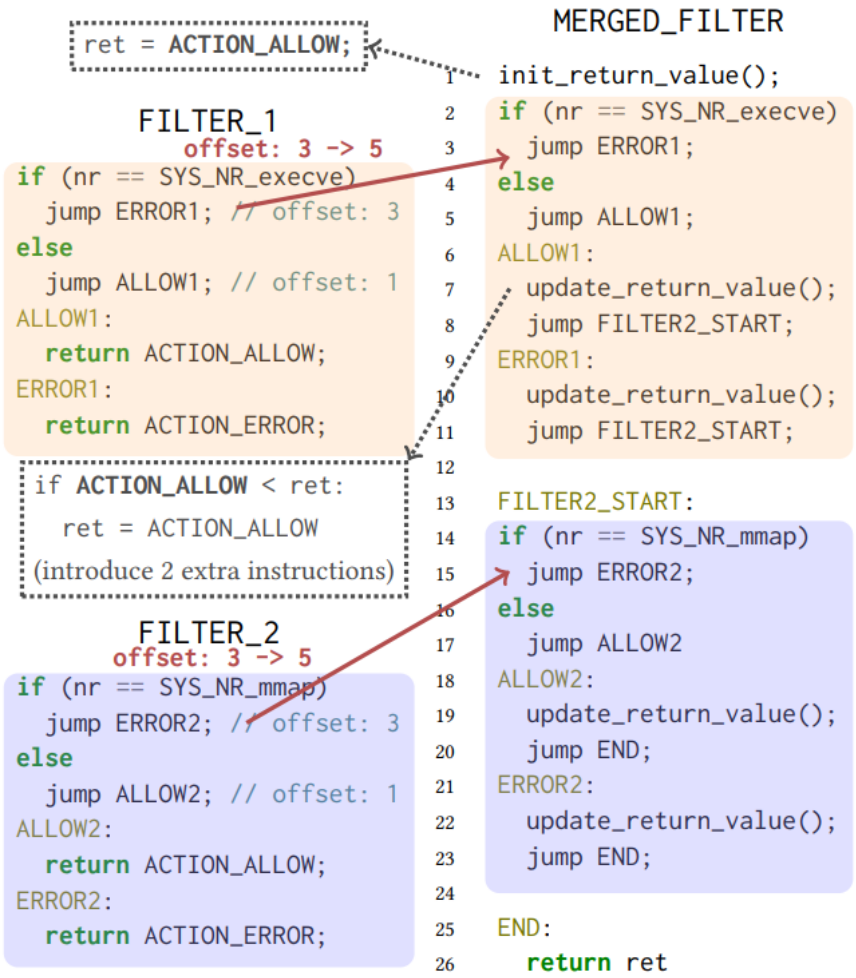


Figure 6.6: KFuse merges two seccomp BPF filters by rewriting the return instructions and updating jump offsets. Dotted boxes: the actual instructions of the conceptual instruction.

6.5.2.3 Recalculating jump offsets.

Jump offsets need to be recalculated based on return instructions because replacing one single return instruction with multiple instructions changes the extension size and misaligns the original jump offsets. KFuse calculates the new offset by counting the number of return instructions between a jump and its destination. For instance, at line 3 in Figure 6.6, the jump offset is updated to 5 from 3 because there is a return instruction between the jump and its destination in the original program, assuming that updating the return value takes 2 additional instructions.

(a) Original programs with tail calls (b) Merged program

Figure 6.7: KFuse rewrites tail calls into direct jumps to merge BPF programs that are connected by tail calls. Note that (a) can be reduced to a DAG because B-C-D is a bounded loop—each node can be treated as a different node.

6.5.3 Optimizing tail calls

A BPF program can tail-call other BPF programs to form a chain (*e.g.*, Prog A, B, C, D and E in Figure 6.7a). To do so, the caller first reads the address of callee from a program array and then performs an indirect jump to the target address. Given the high cost of indirect jumps with retpoline, the JIT compiler, if enabled, will try to optimize the tail call by rewriting the indirect jumps into direct jumps if possible. Note that it is not always possible, *e.g.*, on x86, the target address of the direct jump instruction is stored in a 32-bit signed integer so jumping further than a 32-bit signed integer can only be indirect. Also, this requires to enable JIT, which is not always available or secure [183].

BPF programs can use tail calls to form a bounded call loop by tail-calling back to the caller (*e.g.*, Prog B, C and D in Figure 6.7a). BPF verifier imposes a constraint on the number of tail calls (`MAX_TAIL_CALL_CNT`) to ensure that the program does not execute infinitely by a call loop.

KFuse eliminates the need of a memory access that reads the jump target from the extension array by merging the target extension into the caller. Moreover, KFuse converts the indirect jumps to direct jumps if any. Note that this optimization is done independently from JIT, and is before JIT if it is enabled. Therefore, it is available to architectures without a JIT compiler implementation.

KFuse performs a depth-first search to figure out all target programs when a program that uses tail calls is loaded. KFuse then merges them into one single program and converts the original tail calls to direct jumps to the offset at which the target program is in the merged program, as shown in Figure 6.7b. A backward jump is permitted in the merged program. The merged program still always terminates and is a directed cyclic graph (Section 6.5.5.1), because KFuse enforces the constraint of tail-call count (`MAX_TAIL_CALL_CNT`)—calling re-

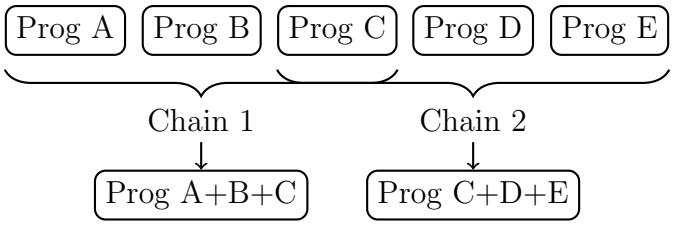


Figure 6.8: Extension is decoupled from the execution.

peated programs can be seen as an edge to a new node (*i.e.*, `MAX_TAIL_CALL_CNT` nodes are allowed at maximum). So, it is equivalent to unrolling the bounded loops.

6.5.4 Transparency to user space

KFuse is fully transparent to user space, by maintaining the original BPF extension structures. It decouples the execution from other operations and only adds a pointer to the fused extensions, as illustrated in Figure 6.8. Therefore, user space APIs, such as looking up, dumping or removing an extension, are not affected. From the user space view, these BPF extensions are still maintained as a chain but only the fused extension is executed when the chain is invoked. On the other hand, the fused extension needs to be regenerated whenever the chain is updated.

KFuse also supports sharing of BPF extensions, when a BPF extension needs to be attached to different chains (*e.g.*, Prog C in Figure 6.8). For example, seccomp BPF filters can be shared by different processes. KFuse merges the shared extension to generate different fused extensions. In my experience, BPF extensions are small in size and the additional memory usage is not concerned, as measured in Section 6.6.6.

6.5.5 Maintaining Safety Properties

Since kernel extension programs are provided from the less privileged user space, the verifier imposes various constraints on them (see Section 6.3.1). The BPF verifier ensures the safety properties listed in Table 6.2 hold, including that programs are directed acyclic graphs (DAG), that programs have no loops/unreachable code and that programs have no invalid memory accesses or jumps. These properties ensure the safety of the kernel when loading user-supplied BPF programs. The verifier imposes a number of constraints to satisfy these properties, also shown in Table 6.2.

Property	Purpose	Maintained
Directed acyclic graph (DAG)	Safety	V
No unreachable code	Safety	V
No invalid memory access	Safety	V
No invalid jump	Safety	V
Constraint		
`MAX_BPF_STACK`	Size	V
`MAX_CALL_FRAMES`	Size	V
`MAX_TAIL_CALL_CNT`	Complexity	V
`BPF_COMPLEXITY_LIMIT_JMP_SEQ`	Complexity	X
`BPF_COMPLEXITY_LIMIT_INSNS`	Complexity	X
`MAX_USED_MAPS`	Size	X
`BPF_MAX_SUBPROGS`	Complexity	X

Table 6.2: **Properties and constraints imposed the verifier.** KFuse preserves all the safety properties.

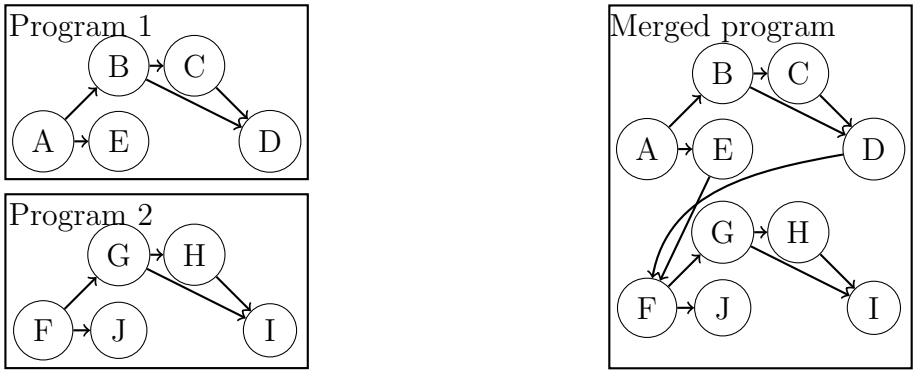
6.5.5.1 Satisfying safety properties

The fused BPF program maintains all safety properties enforced by the verifier on every original BPF program. This can be proven by induction. I sketch the ideas below.

- *Directed acyclic graph (DAG).* Instructions of a BPF program can only be a DAG (no backward jumps or unbounded loops). The DAG property guarantees the termination of the program and also ease the detection of unreachable code. KFuse merges two programs by connecting the exits of one program to the start of the other. The fused BPF program is proven to be a DAG—a graph is a DAG iff it can be topologically ordered; the fused program can be topologically ordered by appending the topological order of the second program to the topological order of the first program.

- *No unreachable code* A BPF program cannot contain unreachable code—it needs to be a connected graph. By connecting the exits of the first program (vertices D and E) to the start of the second program (vertex F), the merged program (Figure 6.9b) is still connected and contains no unreachable code: every vertex in program 1 is reachable from A, every vertex in program 2 is reachable from F, and F is reachable from D or E.

- *No invalid memory access or malformed jumps* A BPF program cannot access invalid memory region (*e.g.*, memory beyond the stack) or include malformed jumps (*e.g.*, backward or out-of-bound jumps). Every program is individually verified before being merged by KFuse. KFuse does not change its memory access and only adds jumps to well-specified destination.



(a) Original programs (b) Merged program

Figure 6.9: KFuse merges two BPF programs (Programs 1 and 2) that have been already verified to be a DAG by connecting the exits of Program 1 (vertices with 0 outdegree, *i.e.*, D and E) to the entry of Program 2 (the vertex with 0 indegree, *i.e.*, F). The merged program is still a DAG.

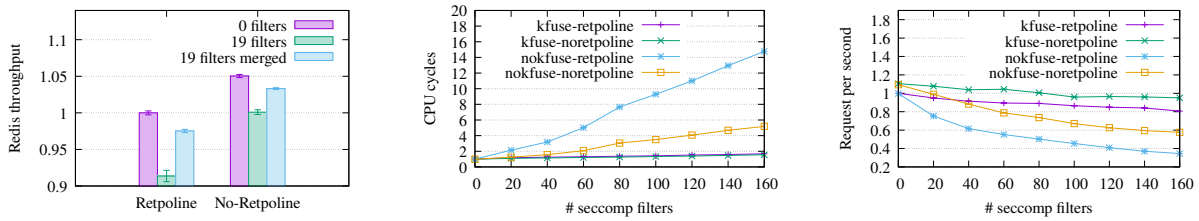
6.5.5.2 Ignoring constraints for bounding verification time.

KFuse is not limited to the constraints imposed by the BPF verifier, if the constraints are solely purposed for bounding the verification time. This is because each individual BPF program is already verified and the fused BPF program does not need to be verified again. The safety properties are maintained by construction.

Table 6.2 shows those constraints used by the verifier. `BPF_COMPLEXITY_LIMIT_JMP_SEQ` and `BPF_COMPLEXITY_LIMIT_INSNS` are used to bound the program size and complexity; `MAX_USED_MAPS` is used to limit the number of maps, and `BPF_MAX_SUBPROGS` is used to limit the subprograms in a BPF program.

6.5.5.3 Domain-specific constraints.

Some kernel subsystems also impose domain-specific constraints. For example, cgroups and tracepoints both limit the number of BPF programs that can be attached to the same hook point by `BPF_CGROUP_MAX_PROGS` and `BPF_TRACE_MAX_PROGS`. These constraints are agnostic to KFuse and KFuse always respect them. This is because the constraints are placed on the original program structure (Section 6.5.4). If the constraints are not satisfied, KFuse will not be invoked. I note that these constraints are not for safety but for performance purposes as they bound the total execution time.



(a) Redis throughput (b) Latency of `getpid` (c) NGINX throughput

Figure 6.10: **(a)** The throughput of Redis with 19 seccomp filters installed by `systemd`. **(b)(c)** The latency of `getpid` and NGINX with different numbers seccomp filters installed for scaling analysis. The results are normalized to baseline, 0 seccomp filters and retpoline enabled.

6.5.6 Implementation

KFuse is implemented in around 500 lines of C code, on top of the Linux kernel (v5.9). As discussed in Section 6.5.1, the main interface is `merge_bpf_progs()`, which takes three input parameters, the pointers of the original BPF programs, and a policy flag that specifies the return value policy, and returns a pointer of the fused program, as shown below.

```
struct bpf_prog* merge_bpf_progs(struct bpf_prog* old_prog, struct bpf_prog*
↪  new_prog, enum return_policy);
```

KFuse uses `BPF_REG_AX` (a special register reserved for the kernel) to hold the temporal return value, which is updated according to the return value policy. The merged program has two-instruction prologue that initializes the return value. Each return instruction is replaced with 5 instructions for the Seccomp return policy and 3 instructions for the tracepoint and cgroup return policy. Finally, the merged program is appended with a two-instruction epilogue which loads the return value from `BPF_REG_AX` to `BPF_REG_A` and returns.

Integrating KFuse. Kernel subsystems can choose to integrate KFuse. The integration is straightforward with a few lines of code. I integrate KFuse in three kernel subsystems, including seccomp, tracepoint, and cgroup. I apply KFuse to seccomp with 20 lines of code (LoC), tracepoint with 28 LoC and cgroup with 34 LoC. I implement a `sysctl` configuration to allow users to enable/disable KFuse.

6.6 EVALUATION

I measure the effectiveness of KFuse in optimizing performance, as well as its overhead in terms of the time it takes for optimization and additional memory cost. To measure

performance, I run performance benchmarks using the three Linux subsystems that integrates KFuse (seccomp, tracepoint and cgroup). All experiments are run on a single server with a 16 core Intel Xeon Silver 4110 CPU at 2.10GHz and 64 GB of memory. I run the experiments in KVM virtual machines. Each VM is configured with 2 VCPUs and 8 GB memory, and the kernel is compiled with the microVM configuration. I use wrk [184] and memtier_benchmark [185] as benchmark clients for NGINX and Redis. I also evaluate the cases where the kernels are compiled without retpoline, which gives a lower bound of the KFuse effectiveness.

Overall, KFuse effectively improves the performance for real-world workloads (7% throughput increase for Redis deployed by `systemd`). The performance benefits of KFuse grows linearly along with the length of chains.

6.6.1 Seccomp

I measure the effectiveness of KFuse in improving the system call performance under seccomp BPF filters. I conduct three experiments:

- I use KFuse to optimize the seccomp BPF filters launched by `systemd` which provides application sandboxing using seccomp. I run Redis and NGINX as real-world applications to assess the benefit of using KFuse on *existing* applications and BPF use cases.
- I conduct a scaling analysis on both system call performance and application performance, with increasing numbers of seccomp BPF filters
- I present the benefit of KFuse for *incremental tightening* which installs seccomp filters during application execution to tighten the system call policy. Incremental tightening is meant to be one of the new BPF use cases that lead to a long chain of BPF extensions and motivate the need to reduce the cost.

6.6.1.1 `Systemd` BPF filters

`Systemd` provides a sandboxing feature that restricts the system calls the sandboxed applications can invoke. I benchmark Redis launched by `systemd`. Redis is shipped on Ubuntu with a `systemd` configuration that installs 19 seccomp BPF filters sandboxing the service including loading kernel modules, gaining new privileges and modifying control groups. Systemd also installs filters for supporting 32-bit systems (*i.e.*, i386). These filters are sized

from 6 instructions to 58 instructions with an average 19.6 instructions. Figure 6.10a shows the throughput of Redis, with the original 19 filters and with the fused filter by KFuse. The results are normalized by the Redis throughput with no BPF filter. I can see that the 19 filters installed by `systemd` lead to non-negligible overhead ($\approx$ 10% throughput decrease). KFuse can significantly reduce the overhead (with only 3%) by speeding up the end-to-end BPF execution time from 957 to 148 nanoseconds (reducing 85%) with the same security policy of original filters.

6.6.1.2 Scaling analysis.

I conduct a scaling analysis to measure the performance overhead of different number of seccomp filters and the benefit of using KFuse. I create a simple BPF filter with 4 instructions: checking if the system call number equals 450, if true, kill the process. Otherwise, allow the call. The system call numbered 450 does not exist, so all the system calls will be allowed.

Figure 6.10b shows the execution time of `getpid` with different numbers of BPF filters. With 20 filters, KFuse can speed up `getpid` performance by 1.8× and 1.2× with and without retpoline. With 160 filters, KFuse can speed up `getpid` performance by 8.8× and 4.9× with and without retpoline.

I also compare the two cases with retpoline disabled (kfuse-noretpoline and nokfuse-retpoline). I find that the execution without the KFuse has 4.7× more instructions, 2.6× more branches and 7.8× more branch misses. The additional instructions and branches come from the loops and branch-misses come from retpoline.

Figure 6.10c shows the throughput of NGINX with different numbers of seccomp filters installed. The throughput drops linearly in both cases with and without retpoline. With 20 filters, KFuse increases throughput by 1.3× with retpoline and 1.1× without. With 160 filters, KFuse increases throughput by 2.3× with retpoline and 1.6× without.

6.6.1.3 Incremental tightening

Incremental system call tightening is a fine-grained temporal system call specialization [159], illustrated in Figure 6.11. It installs multiple BPF filters based on the application's execution phases.

I generate the seccomp BPF filters for different phases of NGINX and Memcached by dynamically tracing their runtime behavior continuously. There are non-deterministic system calls such as those related to timer or signals. I always allow non-deterministic system calls

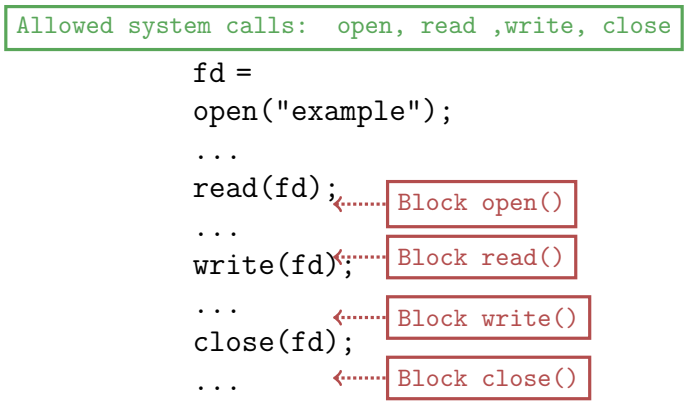
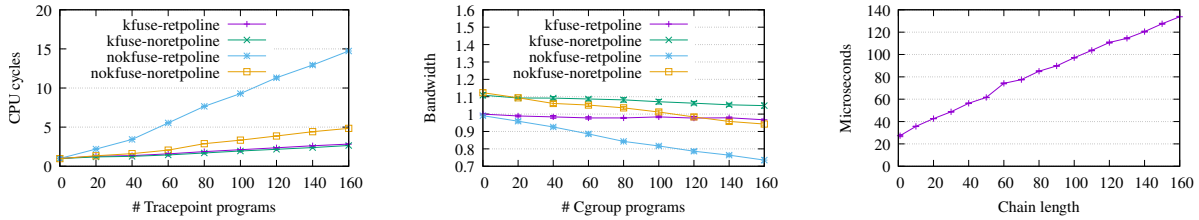


Figure 6.11: Incremental tightening blocks system calls during program execution. This simplified program only uses 4 system calls once, `open`, `read`, `write`, and `close`. The system call is blocked when it will not be again.



(a) Latency of `getpid` (b) Network bandwidth (c) Merging latency

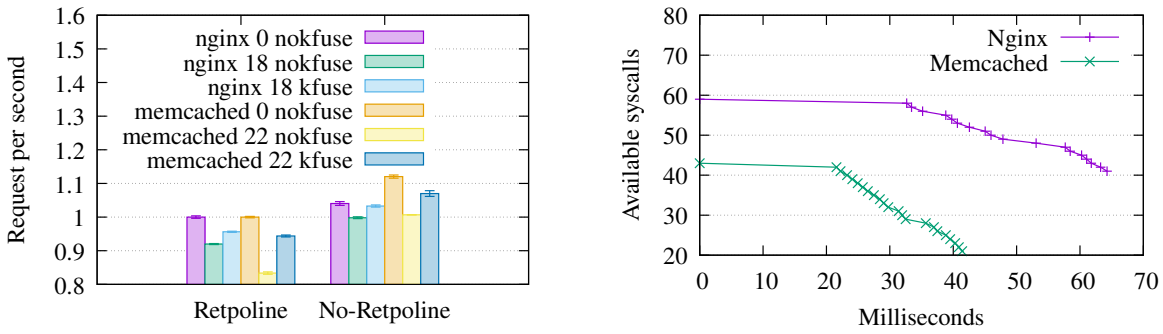
Figure 6.12: Performance measurement of **(a)** `getpid` with different numbers of Tracepoint programs and **(b)** IPerf with different numbers of Cgroup firewall rules. The results are normalized to the baseline, 0 programs and retpoline enabled. **(c)** shows the time KFuse needs to update extensions on a chain.

and only incrementally tighten deterministic system calls. I modify the C library to load the profile and install a seccomp filter when a system call will no longer be needed.

I install 18 seccomp filters for NGINX and and 22 for Memcached. Figure 6.13b shows the number of allowed system calls over time for each application. Figure 6.13a shows that the application performance drops 7%–16% due to the chain of filters. By merging these filters, KFuse eliminates the overheads of calling extensions by 45%–66%.

6.6.2 Tracepoint

Tracepoint enables live kernel debugging and profiling. Multiple BPF programs can be attached to a single tracepoint and all of them are executed when an event is triggered. To experiment with realistic workloads, I use a tracepoint BPF program that not only performs arithmetic instructions but also write data to a BPF map. The program counts the number



(a) Throughput of NGINX and Memcached (b) Number of allowed system call during execution

Figure 6.13: **(a)** Throughput of NGINX and Memcached with 18 and 22 seccomp filters by incremental tightening. The results are normalized to the baseline, 0 seccomp filters and retpoline enabled. **(b)** The number of allowed system calls for NGINX and Memcached over time as incremental tightening blocks system calls during execution.

of times that a system call, `getpid`, is called and write the counter to a map. Figure 6.12a shows the latency of `getpid` with various numbers of programs attached. With 20 programs, KFuse can speed up latency by 1.71× with retpoline and 1.12× without. With 160 programs, KFuse can speed up latency by 5.19× with retpoline and 1.82× without. In the average case of 80 programs, KFuse can speed up latency by 4.09× with retpoline and 1.7× without.

6.6.3 Cgroup

Cgroup provides the mechanism for users to attach multiple BPF programs to implement firewall rules. I attach multiple BPF programs with type `BPF_CGROUP_INET_INGRESS` to experiment with the scenario where multiple firewall rules are applied. The BPF programs simply permit the packet for performance evaluation. I use iPerf3 [186] to measure the TCP bandwidth with various numbers of programs attached. When attaching less than 60 programs, I do not observe obvious performance enhancements. With 60 programs, KFuse improves the bandwidth by 10% with retpoline and 3% without. With 160 programs, KFuse improves the bandwidth by 31% with retpoline and 11% without.

6.6.4 Tail call

To measure KFuse's saving of tail call rewriting, I use the benchmark used by Joly *et al.* [187] for evaluating tail call overhead, in which a BPF program tail-calls various number of other BPF programs and forms chains with different lengths, from 1 to 32. I verify that these tail calls are optimized to direct calls and thus are not slowed down by retpoline. I

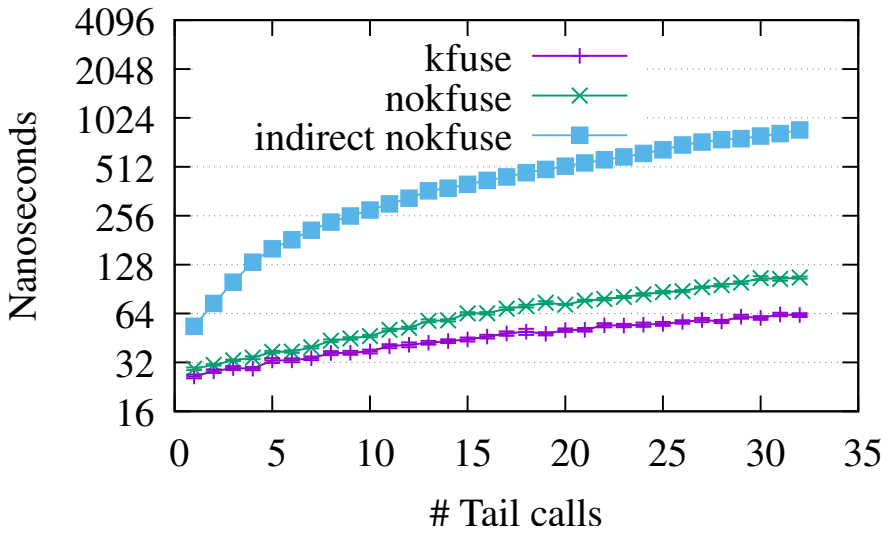


Figure 6.14: Time to call different numbers of tail calls. Tail calls are optimized to direct jumps unless specified.

also evaluate the case when tail calls are not optimized and are indirect jumps. Figure 6.14 shows the average of 50 iterations in three cases. When tail calls are optimized to direct jumps, KFuse saves memory access cost. The average speed-up (16 tail calls) is 1.4× with the minimum (1 tail call) and maximum (32 tail calls) being 1.1× and 1.7×. When tail calls are indirect jumps, KFuse saves both memory access and indirect jump cost. The average speed-up (16 tail calls) is 16.1× with the minimum (1 tail call) being 2× and maximum (32 tail calls) being 32.88×.

6.6.5 Cost of updating a chain

I measure the time KFuse takes to fuse a BPF program chain into an optimized BPF program. KFuse works dynamically by fusing the existing program (the merged chain) with the new program. Specifically, KFuse needs to combine the new program with existing one, reallocate memory for the merged program and generate JITed image for the merged program so the time to merge depends on the length of chain (the size of existing program) and the new program. I measure the time by loading up to 160 programs. As shown in Figure 6.12c, the latency starts from ≈ 27 microseconds when there are only 1 program on the chain and grows to ≈ 138 microseconds when there are 160. When there are 80 programs on the chain, the merging time is ≈ 85 microseconds.

6.6.6 Memory usage

KFuse needs to keep the original BPF bytecode to support sharing of BPF programs and maintain transparency to the user space (Section 6.5.4). This could lead to additional memory overhead. To quantify the overhead, I measure the size of 366 real-world BPF programs from six projects collected by [188] (including ovs, linux, prototype-kernel, suricata, bpf_cilium_test and cilium). The average BPF extension size is merely 4.3 KB, with the median and the 90-th percentile being 0.9 KB and 13.8 KB, respectively. Given the abundant memory of modern computers, the additional memory usage is acceptable.

6.7 CONCLUSION AND DISCUSSION

I demonstrate that multiple BPF programs in a chain can be collectively optimized after each of them is verified. Such optimization can lead to remarkable performance gains. Specifically, when the BPF programs are small in size and the chain is long, the overhead of indirect jumps and loops can be as high as 85% of the end-to-end execution time, leading a 7% application performance loss. I have already seen long chains of BPF extensions in real deployments (*e.g.*, 19 Seccomp BPF filters deployed by `systemd`) and I expect such long chains to be even more common in future use cases, given the explosion of BPF use cases in recent years and the need of fine-grained, domain-specific policies.

I believe that more advanced cross-optimizations for verified BPF programs (such as application-specific optimizations) can be implemented, beyond the basic optimizations currently supported by KFuse. For example, redundant code (*e.g.*, checks [189]) across the BPF extension programs can be removed, in the scope of the entire BPF chain, to further improve the performance; BPF instructions can be reordered across the BPF extension chain (*e.g.*, by placing hot code early in the fused BPF program to save subsequent computation). Those optimizations can be supported in the current KFuse framework. Further, multiple BPF programs at different hooking points are often orchestrated to collectively implement a feature (*e.g.*, socket filters hooked at cgroup and traffic control [190]). Those BPF programs can potentially be optimized together, while maintaining the safety properties.

CHAPTER 7: LIMITATIONS AND EXPERIENCE

In this chapter, I would like to call out the limitations of this thesis and also share the experience I have learned.

7.1 LIMITATIONS

Dynamic analysis such as tracing does not guarantee the result is complete. There are kernel footprint that are not captured. Both MultiK and Cozart use dynamic tracing to profile the parts of a kernel driven by applications. Due to the fundamental limitations of dynamic tracing, these profiles are *not* complete. However, the consequences of incomplete profiles are different for MultiK and Cozart. The missing parts can lead to kernel crashes in MultiK. On the other hand, Cozart debloats kernels at the module level so it is likely to include most of the missing parts. Even when a missing part is actually exercised by an application, the kernel does not crash but returns error codes to the application. This makes Cozart more practical than MultiK.

In addition to incompleteness of tracing profiles, the debloating techniques used by all studies in this thesis, including MultiK, Lupine and Cozart, do not consider non-functional properties that could lead to heisenbugs. An example of a heisenbug could be a kernel module that initializes a variable used by another kernel module and this dependency is not specified. Formal verification can be applied to study if debloating interferes with any such non-functional properties in the system. Besides non-functional properties, this thesis uses empirical methods such as comparing application logs and test coverages to ensure that the debloated kernel has the desired behavior (*i.e.*, to check the functional properties). Such methods provide indirect evidence (instead of guarantees) that we can use to infer that the system exhibits desired behaviors. If the behavior of a Linux kernel can be formally specified, kernels debloated using the methods presented here may also be verified — provide guarantees of desired behaviors and increased confidence to end users.

7.2 EXPERIENCE

In this section, I share experience I learned from the software I have built to explore kernel debloating including Lupine, MultiK and Cozart.

Lupine is a Linux system that is highly specialized. Every kernel module in Lupine is necessary, proven by a manual process. I learned that Linux has the potential to be specialized and beat unikernels in the dimensions including application performance, memory footprint, image size and security. Linux kernel is usually distributed as a general-purpose system. However, applying extensive specialization can turn the kernel into an application-specific system, like unikernels. Specialized Linux kernels are equally lightweight in terms of size and memory footprint. However, in my experience, Linux systems outperform unikernels because there are much more engineering efforts invested in Linux. Linux kernel has various optimization for cache behaviors, scheduling and memory. These optimizations are not trivial. They are the result of decades of time and tremendous efforts from expert developers. Additionally, the applications I benchmark are Linux applications. They are designed and developed to run on Linux systems. Unikernels use translation layers to support them. These translation layers incur overheads. In conclusion, the optimizations and matureness of Linux and its ecosystem (libraries, applications and tools) make it hard to replace Linux.

MultiK is a kernel component that installs and orchestrates specialized kernel code copies. Each application runs on its own, often customized kernel code. I learned that applications can only exercise a small amount of kernel code. However, this does not mean that such specialized "debloated" kernels mean all an application needs. There is a considerable amount of code that are required to maintain the operation of the kernel such as code exercised by kernel threads (*e.g.*, RCU and interrupts). I also learned that applying a custom patch is not a way to build a practical debloating system because this patch is not collectively reviewed by the community and the kernel can be very complex. It is very likely that the patch depends on an assumption that might break other components. For instance, a kernel component might hold RCU lock for too long and cause warnings to the system. Even then such assumption is valid only for the current version. When the kernel is updated, the patch needs to be updated too but the assumption might no longer be valid. Leveraging the existing interface and tools to debloat the kernel is the way that is most likely to succeed.

Cozart takes the lesson from MultiK and uses existing interfaces and tools to debloat, namely KConfig. Cozart is not the first system that uses KConfig. Cozart advances the state-of-the-art by addressing the limitations mentioned in (Section 5.2) of existing frameworks. I recommend that KConfig is the proper interface to debloat a kernel. KConfig provides the confidence that the debloated kernel is stable unlike applying a custom patch. The debloated kernel can also be easily upgraded by via recompilation. The debloated kernel can also be distributed easily by sharing the KConfig file.

CHAPTER 8: CONCLUSION

In this dissertation, I investigate the problem of reducing attack surfaces for contemporary OS kernels via debloating. I raised three key research questions in Chapter 1 and conducted research to answer them.

I answer the first research question – *"can kernel debloating still be useful and effective when comparing to kernels that are built from scratch in a specialized way (e.g., unikernels)?* , by building Lupine which is a minimal Linux system that outperforms unikernels from various dimensions (*e.g.*, boot time, image size, memory footprint and application performance) in Chapter 3. Unikernel systems do not support full POSIX API and they do not perform as well as debloated Linux. More importantly, Linux kernel is supported by numerous expert developers, and tremendous efforts have been spent on optimizing, stabilizing and maintaining it. I conclude that debloated Linux kernels are a better choice in the domain that people presume unikernels are more advantageous.

I answer the second research question – *"what is the preferred way to debloat kernels when considering stability and performance?* , by building MultiK which is a kernel orchestration framework (Chapter 4). Multiple kernels can be specialized for applications and managed by MultiK. I have also build DKut which profiles kernels at basic-block granularity. Integration of MultiK and DKut enables fine-grain kernel debloating. I learned that a large part of the kernel is not used by applications and that aggressive kernel reduction leads instability in the kernel.

These two lessons led to a more practical debloating framework, Cozart (Chapter 5).

I answer the last research question – *"what are the insights and pitfalls to be aware of when developing a kernel debloating framework?* , by studying the state-of-art debloating framework. I built an advanced framework, Cozart. Cozart is a *practical* kernel debloating framework, as it is fully automatic and produces stable kernels. I used Cozart as a vehicle to conduct my study and identified five limitations (Chapter 5). Two of them are essential (the limitations inherent in the nature and assumptions of kernel debloating), and three are accidental (those that exist in current kernel debloating techniques but that are not inherent).

Finally, with the research and experiment results presented in this dissertation, the hypothesis – *reduction of OS kernel bloat ("debloating"), in a practical way, can make operating systems more robust and efficient*, can be validated.

REFERENCES

[1] 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 *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-yiwen pp. 1–13.

[2] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. A. Rabhi, "KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels," in *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., vol. 11050. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-030-00470-5_32 pp. 691–710.

[3] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time OS kernel tailoring," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. [Online]. Available: https://www.ndss-symposium.org/ndss2013/attack-surface-metrics-and-automated-compile-time-os-kernel-tailoring

[4] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," in *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, H. Chen, Z. Zhang, S. Moon, and Y. Zhou, Eds. ACM, 2011. [Online]. Available: https://doi.org/10.1145/2103799.2103805 p. 5.

[5] X. J. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An analysis of performance evolution of linux's core operations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 554–569.

[6] "The rumprun unikernel and toolchain for various platforms," https://github.com/rumpkernel/rumprun, Apr. 2015.

[7] "Javascript library operating system for the cloud," http://runtimejs.org/, Apr. 2015.

[8] D. Williams and R. Koller, "Unikernel monitors: Extending minimalism outside of the box," in *Proc. of USENIX HotCloud*, Denver, CO, June 2016.

[9] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proc. of ACM SoCC*, Carlsbad, CA, Oct. 2018.

[10] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds. ACM, 2013, pp. 461–472.

[11] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[12] H. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker, "MultiK: A Framework for Orchestrating Multiple Specialized Kernels," *CoRR*, vol. abs/1903.06889, 2019. [Online]. Available: http://arxiv.org/abs/1903.06889

[13] H. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 11:1–11:15.

[14] H. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the OS: on the practicality of operating system kernel debloating," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 1, pp. 03:1–03:27, 2020.

[15] K. Chen and H. Shen, "Facechange: Attaining neighbor node anonymity in mobile opportunistic social networks with fine-grained control," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1176–1189, 2017.

[16] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "{SHARD}: Fine-grained kernel specialization with context-aware hardening," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[17] M. Alharthi, H. Hu, H. Moon, and T. Kim, "On the Effectiveness of Kernel Debloating via Compile-time Configuration," in *Proceedings of the 1st Workshop on SoftwAre debLoating And Delayering*, Amsterdam, Netherlands, July 2018.

[18] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "FACE-CHANGE: application-driven dynamic kernel view switching in a virtual machine," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 491–502.

[19] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSV – Optimizing the Operating System for Virtual Machines," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, Philadelphia, PA, USA, June 2014.

[20] J. Corbet, "LWN: A different approach to kernel configuration," https://lwn.net/Articles/733405/, 2018.

[21] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) Than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[22] N. Pitre, "LWN: Shrinking the kernel with link-time garbage collection," https://lwn.net/Articles/741494/, 2018.

[23] N. Pitre, "LWN: Shrinking the kernel with link-time optimization," https://lwn.net/Articles/744507/, 2018.

[24] N. Pitre, "LWN: Shrinking the kernel with a hammer," https://lwn.net/Articles/748198/, 2018.

[25] N. Pitre, "LWN: Shrinking the kernel with an axe," https://lwn.net/Articles/746780/, 2018.

[26] A. Kleen, "Linux Kernel LTO patch set," https://github.com/andikleen/linux-misc, 2014.

[27] V. Adve and W. Dietz, "ALLVM Research Project," https://publish.illinois.edu/allvm-project/ongoing-research/, 2017.

[28] J. Edge, "LWN: Building the kernel with Clang," https://lwn.net/Articles/734071/, 2017.

[29] J. Corbet, "Variable-length arrays and the max() mess," https://lwn.net/Articles/749064/, 2018.

[30] O. R. A. Chick, L. Carata, J. Snee, N. Balakrishnan, and R. Sohan, "Shadow kernels: A general mechanism for kernel specialization in existing operating systems," in *Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys 2015, Tokyo, Japan, July 27-28, 2015*, K. Kono and T. Shinagawa, Eds. ACM, 2015, pp. 1:1–1:7.

[31] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*. USENIX Association, 1986, pp. 93–113.

[32] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: an architecture for reliable device drivers," in *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002*, G. Muller and E. Jul, Eds. ACM, 2002, pp. 102–107.

[33] "Firecracker: Secure and fast microVMs for serverless computing," https://firecracker-microvm.github.io/, retrieved on October 2019.

[34] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A Binary-Compatible Unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*, Providence, Rhode Island, USA, Apr. 2019.

[35] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-process Applications," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, 2014.

[36] t. maeda and a. yonezawa, "kernel mode linux: toward an operating system protected by a type theory," in *advances in computing science – asian 2003. progamming languages and distributed computation programming languages and distributed computation*, v. a. saraswat, Ed. berlin, heidelberg: springer berlin heidelberg, 2003.

[37] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. M. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*. IEEE Computer Society, 2015, pp. 250–257.

[38] "Clive: Removing (most of) the software stack from the cloud," http://lsub.org/ls/clive.html, Apr. 2015.

[39] K. Stengel, F. Schmaus, and R. Kapitza, "EsseOS: Haskell-based Tailored Services for the Cloud," in *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware (ARM'13)*, Beijing, China, Dec. 2013.

[40] "LING," http://erlangonxen.org.

[41] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proc. of USENIX NSDI*, Seattle, WA, Apr. 2014.

[42] D. R. Engler, M. F. Kaashoek, and J. O. Jr, "Exokernel: An Operating System Architecture for Application-level Resource Management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.

[43] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 291–304, 2011.

[44] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "EbbRT: A framework for building per-application library operating systems," in *Proc. of USENIX OSDI*, Savannah, GA, Nov. 2016.

[45] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek, "The Click Modular Router," *ACM Transactions on Compuer Systems*, vol. 18, pp. 263–297, August 2000.

[46] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "Writing User Space Network Drivers," *CoRR*, vol. abs/1901.10664, 2019.

[47] R. Koller and D. Williams, "Will serverless end the dominance of linux in the cloud?" in *Proc. of USENIX HotOS*, Whistler, BC, Canada, May 2017.

[48] "AWS Lambda," https://aws.amazon.com/lambda/, (Accessed on 2016-03-04).

[49] "IBM OpenWhisk," https://developer.ibm.com/open/openwhisk/, (Accessed on 2016-03-04).

[50] "Dockerslim," https://dockersl.im/.

[51] J. Kang, "An Empirical Study of an Advanced Kernel Tailoring Framework," in *The Linux Foundation Open Source Summit*, Vancouver, BC, Canada, Aug. 2018.

[52] J. Kang, "Linux kernel tailoring framework," https://github.com/ultract/linux-kernel-tailoring-framework, Aug. 2018.

[53] "Twistlock — container security & cloud native security," https://www.twistlock.com/.

[54] "Page table isolation (pti)," https://www.kernel.org/doc/html/latest/x86/pti.html.

[55] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[56] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil, and O. Barais, "Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes," INRIA, Tech. Rep. hal-02314830, Oct. 2019.

[57] "Docker hub," https://hub.docker.com/.

[58] "perf - performance analysis tools for linux," http://man7.org/linux/man-pages/man1/perf.1.html.

[59] "On threads, processes and co-processes," https://elinux.org/images/1/1c/Ben-Yossef-GoodBadUgly.pdf.

[60] J. Cormack, "The rump kernel: A tool for driver development and a toolkit for applications."

[61] D. Williams, R. Koller, and B. Lum, "Say goodbye to virtualization for a safer cloud," in *Proc. of USENIX HotCloud*, Boston, MA, July 2018.

[62] "The user-mode linux kernel home page," http://user-mode-linux.sourceforge.net/.

[63] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *Proc. of USENIX HotOS*, Bertinoro, Italy, May 2019.

[64] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[65] "Httpd - apache hypertext transfer protocol server," https://httpd.apache.org/docs/2.4/programs/httpd.html.

[66] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann, "Automatic OS Kernel TCB Reduction by Leveraging Compile-time Configurability," in *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep'12)*, 2012.

[67] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, 1993. [Online]. Available: https://doi.org/10.1145/168619.168635 pp. 203–216.

[68] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: software guards for system address spaces," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, 2006. [Online]. Available: http://www.usenix.org/events/osdi06/tech/erlingsson.html pp. 75–88.

[69] Joint Staff of Washtington, DC, "Information Assurance Through Defense in Depth," Tech. Rep., feb 2000.

[70] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux. conf. au*, 2003.

[71] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor, "Subdomain: Parsimonious server security," in *LISA*, 2000. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa2000/cowan.html pp. 355–368.

[72] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, 2002. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html pp. 17–31.

[73] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of USENIX Annual Technical Conf. (FREENIX Track)*, Anaheim, CA, Apr. 2005.

[74] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, 2013, pp. 139–144.

[75] J. Corbet, "KAISER: hiding the kernel from user space," https://lwn.net/Articles/738975, 2017.

[76] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, 2005. [Online]. Available: https://doi.org/10.1145/1102120.1102165 pp. 340–353.

[77] J. Butler, "DKOM (Direct Kernel Object Manipulation)," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, July 2004.

[78] J. Xiao, H. Huang, and H. Wang, "Kernel data attack is a realistic security threat," in *Security and Privacy in Communication Networks*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds. Springer International Publishing, 2015, pp. 135–154.

[79] 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[80] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[81] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A Study of Feature Scattering in the Linux Kernel," in *IEEE Transactions on Software Engineering (TSE)*, 2018.

[82] 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 11th European Conference on Computer Systems (EuroSys'16)*, 2016.

[83] B. Burns and D. Oppenheimer, "Design Patterns for Container-based Distributed Systems," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, CO, USA, June 2016.

[84] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "Micro" Back in Microservice," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, MA, USA, July 2018.

[85] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," in *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'19)*, Asilomar, California, USA, Jan. 2019.

[86] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, Feb 2019.

[87] N. Savage, "Going Serverless," *Communications of the ACM*, vol. 61, no. 2, pp. 15–16, Feb. 2018.

[88] M. Acher, H. Martin, J. A. Pereira, A. Blouin, D. E. Khelladi, and J.-M. Jézéquel, "Learning From Thousands of Build Failures of Linux Kernel Configurations," INRIA, Tech. Rep. hal-02147012v2f, June 2019.

[89] A. Hubaux, Y. Xiong, and K. Czarnecki, "A User Survey of Configuration Challenges in Linux and eCos," in *Proceedings of 6th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'12)*, Leipzig, Germany, Jan. 2012.

[90] kernel.org, "Kconfig," https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt, 2018.

[91] J. Corbet, "A different approach to kernel configuration," https://lwn.net/Articles/733405/, Sep. 2016.

[92] L. M. Youseff, R. Wolski, and C. Krintz, "Linux Kernel Specialization for Scientific Application Performance," University of California Santa Barbara, Tech. Rep. 2005-29, 2005.

[93] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee, "An Application-Oriented Linux Kernel Customization for Embedded Systems," *Journal of Information Science and Engineering*, vol. 20, no. 6, pp. 1093–1107, 2004.

[94] C.-T. Lee, Z.-W. Hong, and J.-M. Lin, "Linux Kernel Customization for Embedded Systems By Using Call Graph Approach," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, Jan. 2003.

[95] J. Kang, "A Practical Approach of Tailoring Linux Kernel," in *The Linux Foundation Open Source Summit North America*, Los Angeles, CA, Sep. 2017.

[96] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A Robust Approach for Variability Extraction from the Linux Build System," in *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, 2012.

[97] K. Germaschewski and S. Ravnborg, "Kernel configuration and building in Linux 2.5," in *Proceedings of the 2003 Linux Symposium*, Ottawa, Ontario, Canada, July 2003.

[98] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14s)*, Hyderabad, India, 2014.

[99] S. She and T. Berger, "Formal Semantics of the Kconfig Language," Electrical and Computer Engineering, University of Waterloo, Canada, Tech. Rep., Jan. 2010, technical Note.

[100] B. Veer and J. Dallaway, "The eCos component writer's guide," *Available: ecos. sourceware. org/ecos/docs-latest/cdl-guide/cdlguide. html*, 2000.

[101] "Configuring the FreeBSD Kernel," https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig-config.html, 2019.

[102] A. Emmanoulopoulou, "infographic: How many people use Ubuntu?" https://blog.ubuntu.com/2016/04/07/ubuntu-is-everywhere, Apr. 2016.

[103] M. Janakiram, "10 Reasons Why Ubuntu Is Killing It In The Cloud," https://www.forbes.com/sites/janakirammsv/2016/01/12/10-reasons-why-ubuntu-is-killing-it-in-the-cloud, Jan. 2016.

[104] "Amazon Linux 2," https://aws.amazon.com/amazon-linux-2.

[105] L. man page, "addr2line(1)," https://linux.die.net/man/1/addr2line, 2019.

[106] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem," in *Proceedings of the Sixth Conference on Computer Systems (Eurosys'11)*, Apr. 2011.

[107] A. Biere, "Picosat essentials," *JSAT*, vol. 4, pp. 75–97, 2008.

[108] "the cloud market," https://thecloudmarket.com/stats#/by_platform_definition.

[109] "Apache-Test," http://perl.apache.org/Apache-Test/.

[110] "nginx-tests," https://github.com/nginx/nginx-tests.

[111] "Memcached Test," https://github.com/memcached/memcached/tree/master/t.

[112] "The MySQL Test Suite," https://dev.mysql.com/doc/refman/5.7/en/mysql-test-suite.html.

[113] "PHP Test," https://github.com/php/php-src/tree/master/tests.

[114] "Redis Test," https://github.com/antirez/redis/tree/unstable/tests.

[115] "LAMP," http://ampps.com/lamp.

[116] A. Savoia, "Code coverage goal: 80% and no less!" https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html, July 2010.

[117] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code Coverage at Google," in *Proceedings of the 2019 12th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2019)*, 2019.

[118] V. J. Manés, H. Han, C. Han, S. K. Cha, M. Egele, , E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *arXiv:1812.00140*, Apr. 2019.

[119] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, Toronto, Canada, 2018.

[120] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, June 2005.

[121] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, Lisbon, Portugal, Sep. 2005.

[122] C. Cadar and K. Sen, "Symbolic Execution For Software Testing: Three Decades Later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[123] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, Dec. 2008.

[124] "FIASCO: The L4Re Microkernel," http://os.inf.tu-dresden.de/fiasco, retrieved on October 2019.

[125] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, July 2015.

[126] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Bergamo, Italy, Aug. 2015.

[127] T. Xu, V. Pandey, and S. Klemmer, "An HCI View of Configuration Problems," *arXiv:1601.01747*, Jan. 2016.

[128] aobench, "Ambient Occlusion Benchmark," https://github.com/gnzlbg/aobench, 2019.

[129] "SECure COMPuting with filters," https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[130] "Using the linux kernel tracepoints," https://www.kernel.org/doc/html/latest/trace/tracepoints.html, accessed: 09/04/2021.

[131] "BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more," https://github.com/iovisor/bcc, accessed: 09/04/2021.

[132] "Xdp express data path," https://www.iovisor.org/technology/xdp, accessed: 09/04/2021.

[133] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, X. A. Dimitropoulos, A. Dainotti, L. Vanbever, and T. Benson, Eds. ACM, 2018, pp. 54–66.

[134] "Linux system exploration and troubleshooting tool with first class support for containers," https://github.com/draios/sysdig, accessed: 09/04/2021.

[135] C. R. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020.

[136] J. Corbet, "KRSI — the other BPF security module," https://lwn.net/Articles/808048/, 2019, accessed: 09/04/2021.

[137] M. Bachl, J. Fabini, and T. Zseby, "A flow-based IDS using Machine Learning in eBPF," *CoRR*, vol. abs/2102.09980, 2021. [Online]. Available: https://arxiv.org/abs/2102.09980

[138] L. Deri, S. Sabella, and S. Mainardi, "Combining System Visibility and Security Using eBPF," in *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019*, ser. CEUR Workshop Proceedings, vol. 2315. CEUR-WS.org, 2019. [Online]. Available: http://ceur-ws.org/Vol-2315/paper05.pdf

[139] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating Linux Security with eBPF iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. ACM, 2018.

[140] C. Small and M. Seltzer, "VINO: An Integrated Platform for Operating System and Database Research," 1994.

[141] "When ebpf meets fuse," https://events19.linuxfoundation.org/wp-content/uploads/2017/11/When-eBPF-Meets-FUSE-Improving-Performance-of-User-File-Systems-Ashish-Bijlani-Georgia-Tech.pdf, accessed: 09/04/2021.

[142] K. Kourtis, A. Trivedi, and N. Ioannou, "Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF," *CoRR*, vol. abs/2002.11528, 2020. [Online]. Available: https://arxiv.org/abs/2002.11528

[143] Z. Wu, M. Xie, and H. Wang, "Swift: A Fast Dynamic Packet Filter," in *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008. [Online]. Available: http://www.usenix.org/events/nsdi08/tech/full_papers/wu/wu.pdf

[144] "Linux Native, api-aware networking and security for containers," https://cilium.io/, accessed: 09/04/2021.

[145] M. A. M. Vieira, M. S. Castanho, R. Pacifico, E. R. da Silva Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Comput. Surv.*, vol. 53, no. 1, 2020.

[146] Y. Choe, J. Shin, S. Lee, and J. Kim, "eBPF/XDP Based Network Traffic Visualization and DoS Mitigation for Intelligent Service Protection," in *Advances in Internet, Data and Web Technologies, The 8th International Conference on Emerging Internet, Data and Web Technologies, EIDWT 2020, Kitakyushu, Japan. 24-26 February 2020*, ser. Lecture Notes on Data Engineering and Communications Technologies, vol. 47. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-39746-3_47

[147] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *26th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2020, Hong Kong, December 2-4, 2020*. IEEE, 2020.

[148] J. Hong, S. Jeong, J. Yoo, and J. W. Hong, "Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring," in *14th International Conference on Network and Service Management, CNSM 2018, Rome, Italy, November 5-9, 2018*. IEEE Computer Society, 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8584971

[149] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging eBPF for programmable network functions with IPv6 segment routing," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*. ACM, 2018.

[150] S. Baidya, Y. Chen, and M. Levorato, "eBPF-based content and computation-aware communication for real-time edge computing," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018*. IEEE, 2018.

[151] K. Suo, Y. Zhao, W. Chen, and J. Rao, "Demo/poster abstract: Efficient and flexible packet tracing for virtualized networks using eBPF," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018*. IEEE, 2018.

[152] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," in *30th International Teletraffic Congress, ITC 2018, Vienna, Austria, September 3-7, 2018 - Volume 1*. IEEE, 2018.

[153] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman, "Synthesizing Safe and Efficient Kernel Extensions for Packet Processing," *CoRR*, vol. abs/2103.00022, 2021. [Online]. Available: https://arxiv.org/abs/2103.00022

[154] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture," in *Proceedings of the ACM SIGCOMM 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 1999, Cambridge, Massachusetts, USA*. ACM, 1999.

[155] "[PATCH net-next 8/9] net: filter: rework/optimize internal BPF interpreter's instruction set," https://lore.kernel.org/netdev/1395404418-25376-9-git-send-email-dborkman@redhat.com/, accessed: 09/04/2021.

[156] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 2013. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim

[157] "Performance Benchmark Analysis of Egress Filtering on Linux," https://kinvolk.io/blog/2020/09/performance-benchmark-analysis-of-egress-filtering-on-linux/, accessed: 09/04/2021.

[158] "cgroups(7) — linux manual page," https://man7.org/linux/man-pages/man7/cgroups.7.html, accessed: 09/04/2021.

[159] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[160] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-Phase Execution of Application Containers," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10327. Springer, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_11

[161] "new seccomp mode aims to improve performance," http://kernsec.org/pipermail/linux-security-module-archive/2020-June/020706.html, accessed: 09/04/2021.

[162] "Linux Socket Filtering aka Berkeley Packet Filter (BPF)," https://www.kernel.org/doc/Documentation/networking/filter.txt, accessed: 09/04/2021.

[163] 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," 2019.

[164] "Retpoline: A branch target injection mitigation," https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, accessed: 09/04/2021.

[165] "Evaluation of tail call costs in ebpf," https://www.linuxplumbersconf.org/event/7/contributions/676/attachments/512/1000/paper.pdf, accessed: 09/04/2021.

[166] "systemd," https://www.freedesktop.org/wiki/Software/systemd/, accessed: 09/04/2021.

[167] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 1993. [Online]. Available: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

[168] "Lsm bpf programs," https://www.kernel.org/doc/html/latest/bpf/bpf_lsm.html, accessed: 09/04/2021.

[169] "Beautifying syscall args in 'perf trace'." http://vger.kernel.org/~acme/perf/linuxdev-br-2018-perf-trace-eBPF/, accessed: 09/04/2021.

[170] Y. Zhong, H. Wang, Y. J. Wu, A. Cidon, R. Stutsman, A. Tai, and J. Yang, "BPF for storage: an exokernel-inspired approach," in *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*. ACM, 2021.

[171] N. Amit and M. Wei, "The Design and Implementation of Hyperupcalls," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/amit

[172] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018.

[173] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler, "LBM: A Security Framework for Peripherals within the Linux Kernel," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019.

[174] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, B. Christensen, A. Gartrell, M. Khutornenko, S. Kulkarni, M. Pawlowski, T. Pelkonen, A. Rodrigues, R. Tibrewal, V. Venkatesan, and P. Zhang, "Twine: A unified cluster management system for shared infrastructure," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/tang

[175] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: reading kernel memory from user space," *Commun. ACM*, vol. 63, no. 6, 2020.

[176] C. Young, N. C. Gloy, and M. D. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*. ACM, 1995.

[177] N. Amit, F. Jacobs, and M. Wei, "JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/amit

[178] "net: mitigate retpoline overhead," https://lwn.net/Articles/773985/, accessed: 09/04/2021.

[179] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020.

[180] "The LLVM Compiler Infrastructure," https://llvm.org/, accessed: 09/04/2021.

[181] "Binary Optimization and Layout Tool - A linux command-line utility used for optimizing performance of binaries ," https://github.com/facebookincubator/BOLT, accessed: 09/04/2021.

[182] "Xdp performance regression due to config_retpoline spectre v2," http://lkml.iu.edu/hypermail/linux/kernel/1804.1/05171.html, accessed: 09/04/2021.

[183] E. Reshetova, F. Bonazzi, and N. Asokan, "Randomization can't stop bpf jit spray," in *International Conference on Network and System Security*. Springer, 2017.

[184] "Modern http benchmarking tool," https://github.com/wg/wrk, accessed: 09/04/2021.

[185] "Nosql redis and memcache traffic generation and benchmarking tool." https://github.com/RedisLabs/memtier_benchmark, accessed: 09/04/2021.

[186] "iPerf - The ultimate speed test tool for TCP, udp and sctp," https://iperf.fr/iperf-doc.php, accessed: 09/04/2021.

[187] C. Joly and F. Serman, "Evaluation of tail call costs in eBPF," *Linux Plumbers Conference 2020*, 2020. [Online]. Available: https://linuxplumbersconf.org/event/7/contributions/676/attachments/512/1000/paper.pdf

[188] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted Linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019.

[189] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/zhang pp. 479–494.

[190] "Bpf map tracing: Hot updates of stateful programs," https://linuxplumbersconf.org/event/11/contributions/942/, accessed: 09/04/2021.