BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

DYNAMIC PRIVILEGE

by

THOMAS UNGER

B.A., Boston University, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2024

Approved by

First Reader

Jonathan Appavoo, Ph.D.
Associate Professor of Computer Science

Second Reader

Orran Krieger, Ph.D.
Professor of Electrical & Computer Engineering

Third Reader

Renato Mancuso, Ph.D.
Assistant Professor of Computer Science

Fourth Reader

Vasiliki Kalavri, Ph.D.
Assistant Professor of Computer Science

Be content with what you have;
rejoice in the way things are.
When you realize there is nothing lacking,
the whole world belongs to you. Lao Tzu

Acknowledgments

Thank you to my advisor, Jonathan Appavoo, and to my effective co-advisor Orran Krieger. I also thank my committee members, Renato Mancuso and Vasia Kalavri.

To my collaborators: Jim Cadden, Ali Raza, Arlo Albelli, Yara Awad, Han Dong, Albert Slepak, Ryan Sullivan, Eric Munson, Parul Sohal, Amos Waterland, Ting Hsu, Matt Boyd, and Ke Li.

To my labmates and friends: Dan Schatzberg, Andrea Burns, Lance Galletti, Vitali Petsiuk, Wil Koch, Aanchal Malhotra, Nikolaj Volgushev, Sahil Tikale, Ugur Kaynar, (little) Ali Raza, and Sanskriti Sharma.

To my Red Hat mentors and friends: Larry Woodman, Richard Jones, Heidi Dempsey, Uli Drepper, and Daniel Bristot de Oliveira.

Thank you to Luke Massa for many interesting conversations about the work, Katherine Chase for mental engineering, and Adam DiBattista for proofreading.

To the BU Math/CS meditation practice group, especially Gregg Harbaugh, Kevin Schattenkirk, Haviland Wright, and Ben Polletta.

Finally, thank you to all my family and friends; you know who you are; I love you.

–Tommy

DYNAMIC PRIVILEGE

THOMAS UNGER

Boston University, Graduate School of Arts and Sciences, 2024

Major Professor: Jonathan Appavoo, PhD
Associate Professor of Computer Science

ABSTRACT

This dissertation introduces *Dynamic Privilege*, an operating system (OS) model. With Dynamic Privilege, threads may transition independently between hardware modes of execution, adding a temporal dimension to hardware privilege beyond the conventional separation in systems software. Dynamic Privilege enables multiple derivative privilege policies and system organizations. Dynamic Privilege differentiates from existing privilege transfers, such as system calls, interrupts, and their associated returns, because it decouples hardware mode transitions from transitions in control flow.

In OS design, developers operate under an implicit assumption: access to hardware privilege is a *static* property. A consistent separation is upheld throughout the entire software lifetime, from the initial development of source code to the creation of binaries and, eventually, the deployment of running systems. This separation distinguishes between code designed to operate with hardware privileges, like the kernel and its modules, and application code that does not possess these privileges.

The ubiquitous use of address spaces and virtual address translation further reinforces this divide, leading to system designs that fundamentally rely on isolating kernels and applications. This separation is explored and manipulated differently in traditional kernel architectures like Monolithic, Microkernels, Exokernels, and Uniker-

nels. These models navigate this design space by statically positioning system components in privileged or unprivileged binaries and attendant address spaces, reaching different trade-offs in performance, fault tolerance, verification, etc.

Kernels are powerful but austere; applications are full-featured but sandboxed. This exploration of Dynamic Privilege focuses on extending user processes with access to hardware privilege, targeting the simultaneous benefits of these two worlds and blurring the conventional distinction between kernel and application.

While we imagine a future where hardware implements Dynamic Privilege directly, this dissertation develops and explores Dynamic Privilege within the Linux OS with a new system call: *kElevate*. kElevate toggles a thread's execution mode, providing full hardware access. With support from the kernel module subsystem, privileged applications build up to OS-level access. They can reuse and re-implement all the kernel symbols, types, macros, inline functions, and standalone subroutines. We pioneer dynamically linking the Linux kernel into privileged processes using standard shared library formats. In this fashion, *elevated processes* may cooperate with the existing kernel or arbitrarily diverge from its design.

We use kElevate to optimize energy and performance across the application-kernel divide while preserving compatibility with existing software. This includes transparent optimization, compatibility with the application binary interface (ABI), dynamic linking, scripting, containerization, and high-level (even interpreted) languages. We demonstrate this on microbenchmarks and the Redis key-value store. The kElevate mechanism is prototyped as a system call for x86_64 and Arm64 processors.

Contents

List of Tables

List of Figures

List of Abbreviations

ABI	.....	Application-Binary Interface
AO	.....	Anticipatory Optimization
ARM	.....	Advanced RISC Machines
ASC	.....	Automatically Scalable Computation
BSD	.....	Berkeley Software Distribution
CPU	.....	Central Processing Unit
DPDK	.....	Data Plane Development Kit
ELF	.....	Executable and Linkable Format
FaaS	.....	Functions as a Service
FUSE	.....	FileSystem in Userspace
GCC	.....	GNU Compiler Collection
IDT	.....	Interrupt Descriptor Table
IDTR	.....	Interrupt Descriptor Table Register
IoC	.....	Inversion of Control
IPC	.....	Instructions Per Cycle
IRET	.....	Interrupt Return
ISA	.....	Instruction Set Architecture
IST	.....	Interrupt Stack Table
KASLR	.....	Kernel Address Space Layout Randomization
KML	.....	Kernel Mode Linux
KVM	.....	Kernel Virtual Machine
LSM	.....	Linux Security Module
MMU	.....	Memory Management Unit
NIC	.....	Network Interface Controller
NSA	.....	National Security Agency
NVRAM	.....	Non-Volatile Random-Access Memory
OS	.....	Operating System
PIC	.....	Position Independent Code
PLT	.....	Procedure Linkage Table
POSIX	.....	Portable Operating System Interface
QEMU	.....	Quick EMUlator
RISC-V	.....	Reduced Instruction Set Computer
ROP	.....	Return Oriented Programming
RTOS	.....	Real-Time Operating System

SDK	.....	Software Development Kit
SELinux	.....	Security-Enhanced Linux
SEUSS	.....	Serverless Execution with Unikernel SnapShots
SMAP	.....	Supervisor Mode Access Prevention
SMEP	.....	Supervisor Mode Execution Prevention
SMP	.....	Symmetric Multiprocessing
SV	.....	State Vector
TCB	.....	Thread Control Block
TLB	.....	Translation Lookaside Buffer
UKL	.....	Unikernel Linux
vDSO	.....	virtual Dynamic Shared Object
VM	.....	Virtual Machine

Chapter 1

Introduction

1 Terminology

We offer a brief glossary of terms used throughout the dissertation:

Term	Definition
Privilege	A separation in processor execution modes at the hardware level, where the privileged mode can access a superset of system memory and instructions. Primarily used to limit and isolate user environments from a more powerful OS monitor. See Chapter 1 Section 1.3.
Static Privilege	Traditional operating system model where a kernel executable always runs with privilege, while user processes always run without. See Chapter 3 Section 3.1.1.
Dynamic Privilege	An OS model described in this dissertation where threads can independently transfer between privilege levels. A generalization of the static model of privilege. See Chapter 3 Section 3.1.1.
To Elevate/ Lower	Verbs describing the transition of a thread to a more/less privileged state.
Elevated Code	A sequence of instructions running with privilege via Dynamic Privilege.
Elevated Thread Elevated Process	Similarly, a thread/process running with privilege. Chapter 1 Section 1.2
Source-to-Runtime Spectrum	A container term, collecting multiple stages during the lifetime of software. It captures software design from source code to post-compilation built binaries to runtime execution. Section 2.2 Section 2.2.2

Table 1.1: Definitions of terms used in the dissertation.

2 Dissertation Overview

This dissertation challenges a core assumption made by all Operating Systems (OSs): *there should be a static divide between code that runs with hardware privilege and code that runs without.* Privileged code (typically found in OS kernels and modules) is powerful in its ability to affect any aspect of the system. Still, it lacks the rich software ecosystem, the hallmark of user-level software. This work explores the marriage of these worlds by enabling user-level threads to take on privilege with the full power of supervisor mode, unlocking unmitigated access to hardware. We call this ability Dynamic Privilege, emphasizing that threads can dynamically adjust their privilege level at runtime. This new ability has implications for the design and implementation of the complete software stack.

Threads utilizing Dynamic Privilege gain hardware privilege and, thus, unrestricted hardware access, including access to all memory and the entire instruction set. We demonstrate that this low-level hardware access is sufficient for building up to OS-level "access," where all modern OS macros, types, data structures, and subroutines become available. OS-level access allows this empowered software to exploit privilege with newly developed code and reuse the kernel's internals like a Software Development Kit (SDK).

Dynamic Privilege transforms the kernel binary and source into a practical toolbox for "elevated processes", processes using Dynamic Privilege. It enables two key functionalities: firstly, it allows elevated processes to dynamically link to the kernel instance at startup as if it were a regular shared library. Secondly, it provides the ability to incorporate existing kernel source into the development and compilation of applications or libraries with elevated privileges. This utility simplifies the development process, whether invoking existing routines in the running kernel or crafting new hybrid code paths based on the kernel source code.

Today, the classic approach to exploiting the kernel in novel ways is through some form of OS extension. OS extension is typically done by downloading application code into the kernel.[1] Running elevated processes sidesteps two core limitations of code downloading. First, developers need not develop new code that conforms to the kernel's restricted execution model[2], see Chapter 3 Section 3.3. Second, elevated processes maintain direct procedural control over their execution, unlike the inversion of control suffered by extensions (see Chapter 3 Section 3.3.1), which merely register handlers within the kernel to be called at the OS's convenience. In Chapter 3 Section 3.3, we provide a three-part conceptual framework for understanding Dynamic Privilege in relation to processes, modules, and alternative kernel designs.

Dynamic Privilege provides fine-grained access to privilege along two dimensions. First, elevated processes can access hardware privilege on a per-thread granularity. Second, threads can toggle privilege access on and off for arbitrary periods. Fine-grained access to privilege enables incremental approaches to working with application software, generally designed to run in user mode and may require some system adaptation to run with privilege see Chapter 5. This offers an advantage over the other OS architecture that only runs application code with privilege, Unikernels (see Section 2.2 Section 2.2.3). Dynamic Privilege offers a generalization of the Unikernel because it can support multiple elevated processes and can support the `fork()` system call.

We explore the model of Dynamic Privilege within the Linux OS by adding a new system call, "kElevate". While we defer discussion of kElevate to Chapter 3, here we highlight that when discussing it more generally as an OS model, we use the term "Dynamic Privilege", and when referring to our mechanism which facilitates the

[1] This encompasses a wide range of approaches, including modules, transactions, formal methods, and object-oriented techniques.

[2] Consider calling a kernel function from an application context that exploits a high-level language like C++ and the POSIX interface.

model, we will use the name of the system call, "kElevate."

As part of our experimental exploration in Section 4.2, we demonstrate significant performance and energy optimization using Dynamic Privilege while retaining the strongest form of software compatibility, complete application-binary interface (ABI) compatibility. We show libraries and tools that use Dynamic Privilege to optimize applications transparently. Specifically, by integrating kElevate, a system call, into standard application libraries, we utilize dynamic linking[3] to enhance the performance of existing binaries without needing to recompile or modify them. We demonstrate that standard dynamic linking techniques are effective. Interposing new code paths between an application and its dependent libraries like libc, the C standard library, allows us to create privileged code paths enabling applications to perform tasks not originally designed for, such as directly invoking internal code within the running kernel.

Furnishing user threads with access to hardware privilege naturally raises security concerns. To address this, we treat access to Dynamic Privilege as a protected ability. To this end, our Linux implementation gates access to the kElevate mechanism using the existing credential system. The credential system is the same security subsystem that is used to prevent unauthorized users from, for instance, inserting rogue kernel modules. Beyond this, we believe that unrestricted use of Dynamic Privilege has a place in the context of cloud-hosted computation. In particular, one usage context that obviates security concerns is the one adopted by Unikernels, which assumes that a single user (or provider) controls an entire physical node (or virtual machine). The elevated process(es) become part of the trusted compute base. Single provider datacenters, ever-cheaper hardware, and the proliferation of virtualization hardware argue in favor of the continued relevance of this scenario. Section 1.6 of this chapter

[3]Dynamic linking is a standard practice for loading and linking an application to libraries. It joins code paths at runtime, not at compile time. See Chapter 5.

discusses security and Dynamic privilege in greater detail.

As part of our exploration of Dynamic Privilege, we consider several use cases. Chapter 4 offers examples demonstrating access to low-level capabilities like bracketing privilege access in time, as well as accessing privileged memory, registers, and hardware data structures. It also shows OS-level examples, including executing kernel subroutines and utilizing kernel data, types, macros, and dynamic linking to kernel symbols. We explore OS-level access from full-featured application contexts using Python interpreters, Jupyter notebooks, and data-science libraries. In Chapter 5, we discuss the use of adaptors, a mechanism for mitigating issues that arise when user code executes with hardware privilege. Finally, we demonstrate Dynamic Privilege's application to optimization via "shortcutting" in Section 4.2.

Dynamic Privilege distills access to privilege as an independent variable, and yet it can be easily integrated into an existing static privilege OS as demonstrated by our exploration of kElevate in Linux. We implement Dynamic Privilege as a system call, kElevate, on x86 and ARM architectures using 100s of lines of code. As such, we provide evidence that support for Dynamic Privilege can be retrofitted into a modern general-purpose OS, Linux. While we have not done so, we believe that integration into a restricted or specialized OS should be, if anything, more straightforward.

2.1 Thesis Statement:

We introduce Dynamic Privilege, an OS model enabling threads to adjust their hardware privilege levels on the fly, challenging the traditional separation between kernel and application. Dynamic Privilege provides granular control and requires only minor modifications to support it in an existing OS. Further, Dynamic Privilege enables new system evolution and optimization while maintaining compatibility with existing software ecosystems.

The following section, Section 1.3, provides an expanded discussion of privilege to ensure the reader has the necessary context and background before proceeding. The introduction then proceeds with three sections. The first, Section 1.4, presents Dynamic Privilege in terms of three properties: 1) control, 2) access, and 3) execution modes. The second, Section 1.5, summarizes the core contributions of this dissertation. Section 1.6 provides the interested reader with an expanded security discussion due to the inherent relationship to security. Finally, Section 1.7 concludes the introduction with an overview of the rest of the dissertation.

3 Privilege

Privilege is the fundamental organizational principle of operating system design. OS architectures are defined by which functionalities are granted privileged status and are carefully isolated from non-privileged components. We commonly refer to the privileged side of the divide as the OS's "kernel."

The division between privileged and non-privileged software is evident in every stage of the software lifecycle, from separate source code and build systems to independent binaries and hardware-enforced runtime isolation. See Figure 1·1 for an introduction to what we call the "Source-to-Runtime Spectrum." The choice of implementing specific functionality within the kernel is not merely conventional, it determines the system's properties, such as the limits of application optimization (Engler et al., 1995a), fault tolerance properties (Accetta et al., 1986), and the kernel's eligibility for formal verification (Heiser and Elphinstone, 2016).

Computing hardware assists the OS in realizing privilege separation at runtime. It provides modes and instructions for constructing privileged domains and enforces separation between them. Hardware privilege is the basis for security and isolation, ensuring that applications cannot directly access system resources. Instead, applica-

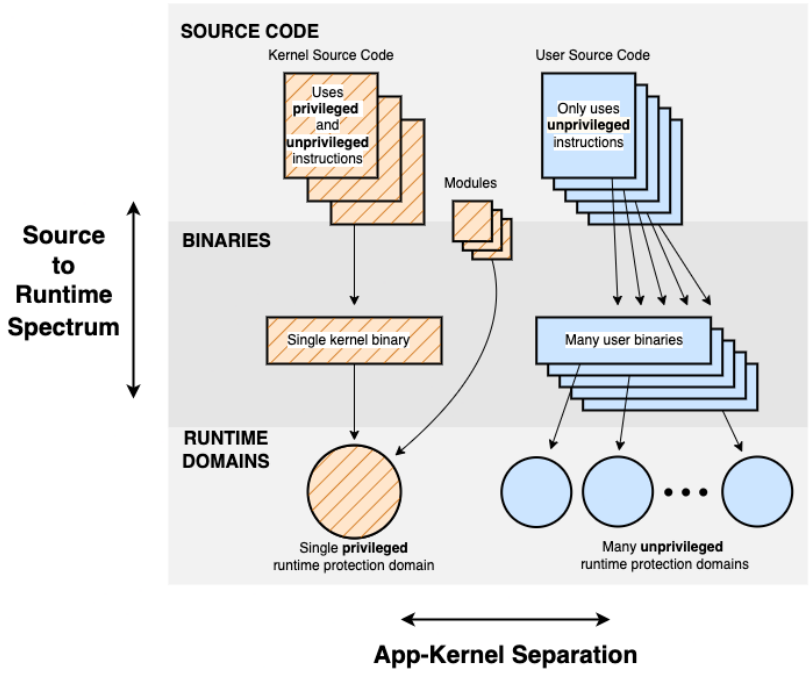

Figure 1·1: We refer to this model as the Source-to-Runtime Spectrum. We use it to analyze the application-kernel separation throughout the software lifecycle. Segregated OS and application source codebases compile into separate executables (binaries). On UNIX-like systems, these exist as separate ELF files. These binaries are loaded and booted as separate runtime domains. These runtime domains, namely the kernel and user runtime domains, are executed in secluded corners of the same address space or separate address spaces entirely, depending on the OS. At runtime on Linux, the Memory Management Unit (MMU) uses privilege to separate applications and the kernel at runtime.

tions make requests mediated by the kernel. General-purpose computers use hardware privilege separation, including data center servers, laptops, smartphones, and other devices.

All existing OSs that we are aware of establish a *static* notion of the types of functionality realized on either side of the privilege boundary.[4]

Traditionally, there was only a single layer of privilege separation between the application and OS. Engineers added another layer to support virtualization, the hypervisor. Hypervisors typically provide more robust isolation than OSs. While this dissertation focuses on the two primary modes utilized by common general-purpose OSs–user and supervisor–it would be interesting to consider extending this mechanism to other privilege modes like hypervisor, monitor, and system management modes in future work.

While existing OSs utilize a static model of privilege, researchers have done significant work under the banner of "customizable OSs" to facilitate a dynamic model of kernel functionality within their static privilege constraints. These OSs provide mechanisms for reimplementing or injecting application functionality into the kernel. Various approaches span dynamically linked modules (Kernel-Contributers, 2022), hardware transactions (Small and Seltzer, 1994), object-oriented OSs (Krieger et al., 2006; Campbell and Tan, 1995), and typesafe languages (Bershad et al., 1995). These extension mechanisms retain the static separation of privileged versus unprivileged code. They prevent application control and operate under restricted execution models, as discussed in the following section.

[4]This is true for alternative OS architectures such as Monoliths, Microkernels, Exokernels, libraryOSs (Unikernels), and single address space OSs which all follow this static design.

4 Control, Access, and Execution-Models

Dynamic Privilege blurs the traditional lines between privileged and non-privileged functionality. When implemented within a general-purpose OS, it allows for modifying core system structure from the high-level, familiar, and easy-to-use application context. To structure our discussion of Dynamic Privilege, the goals of our effort, and its relation to existing research, we provide the following framework, which unites three core properties: application control, OS-level access, and the application execution model. After briefly describing each, we discuss why Dynamic Privilege is critical to enabling us to achieve these three properties.

- **Application Control:** The application determines the execution flow and decides when to call methods procedurally. By contrast, the framework extension paradigm enforces an "Inversion of Control" (Gamma et al., 1994) by determining when and under what conditions a user handler will execute.

- **Kernel-Access:** The ability to locate and modify kernel software. Kernel access implies decomposing and rebuilding abstractions to meet application functionality and optimization targets.

- **Application Execution Model:** Applications have access to rich software ecosystems, language and library support, threading, scripting, and run in user mode. Developers restrict kernels by design. Kernels typically provide primitive libraries, baroque build systems, and limited threading support.

4.1 Putting Together Kernel-Access, Control, and Application Model

Dynamic Privilege can empower applications with OS-level access. Kernel access allows application and library software to optimize, redesign, and implement core OS structures. Our system uses kElevate to provide arbitrary hardware access. From

	Linux Process	Extension	UKL	Process w/ kElevate
App. Control	✓	X	✓	✓
Kernel-Access	X	✓	✓	✓
App. Exec. Model	✓	X	∼	✓

Table 1.2: Comparison between Linux Processes, classical OS extension techniques, a modern Unikernel: UKL, and a Linux process with access to hardware privilege via kElevate. Linux applications have procedural control of their execution and access to the rich software ecosystem conferred by the application execution model but no OS-level access. Classical OS extensions rely on registering handlers into the OS framework, inverting control, and accepting the restricted kernel model. UKL buys back control but compromises between the application and kernel execution models. kElevate claims all three.

there, we build up to the same type of OS-level access one would expect when writing core kernel code or a module. Unlike existing extension models, we demonstrate that Dynamic Privilege allows elevated processes to retain direct control of their procedural execution. Uses of the kElevate mechanism simultaneously achieve all three of these properties.

5 Core Contributions

This dissertation identifies a coupling in all operating systems that intrinsically links software codebase (kernel, application) to modes of hardware privilege (supervisor, user). We hypothesized that introducing the dynamic ability to change privilege could address systems' challenges. In exploring this hypothesis, we made the following contributions:

- Defined the OS model of Dynamic Privilege, introduced the notion of fine-grained hardware privilege switching.
- Prototyped a mechanism for extending an existing commodity OS kernel with support for Dynamic Privilege by adding a new system call, kElevate, with some modest kernel changes. We support this for x86_64 and Arm64 processors with a few hundred lines of code.

- Produced a Dynamic Privilege library, which codifies our approach to using kElevate and facilitates future mechanism use. It offers us the flexibility to modify kernel data structures, address issues that emerge from executing application threads in privilege, and formalize our methodologies, offering hooks for future extension.

- Explored the use of Dynamic Privilege through a set of case studies, see Chapter 4.

- Use the Dynamic Privilege library to enact application optimization and quantify the results in the context of Linux. Using this library, we implement shortcuts, see Section 4.2.2 to reduce latency and energy cost on core system IO paths. We quantify these improvements on microbenchmarks and real-world applications like Redis.

6 Security

At first glance, allowing a user-level process direct access to hardware privilege may be considered anathema. Preventing privilege escalation is a first-order concern for every general-purpose system; major operating systems have researched and adopted fine-grained credential systems. The thought of intentionally providing users with direct access to privilege seems to subvert our most basic expectation of how to construct systems.

Providing ultimate privilege to applications naturally raises security concerns. Here, we consider an existing all-powerful kernel mechanism, dynamically loadable kernel modules. While the kernel uses capabilities to secure the module insertion mechanism, it does not attempt to verify the module's integrity; this is left entirely to secondary vetting. Instead of subverting existing capability systems, we argue that Dynamic Privilege composes with and complements them.

We use the Linux kernel's capability system to protect access to the kElevate system call. We do not prove any security properties of elevated code. Developers can use secondary techniques like code review, testing, static and dynamic analysis, and signing to produce and distribute secure software if they wish. Many OSs use fine-grained capabilities to provide nuanced control over privileged resources. While the most naive use of kElevate would be to grant unrestricted privilege to a process, one could instead mark individual restricted elevated code segments with capabilities allowing for their controlled use. This approach combines the flexibility of Dynamic Privilege with the nuanced security of capabilities.

To conclude our discussion of security, let us consider scenarios in which security is not required at the OS level. Unikernel researchers tend to dedicate an entire physical node or virtual machine to a single application. With co-running applications eliminated, the threat model reduces to a single trusted application. Unikernels provide great latitude to specialize systems on a per-application basis. Developers can use elevated processes in the same context, dedicating hardware. Three hardware trends support the continued relevance of this context: first, Moore's law provides ever denser (and thus cheaper) hardware; second, Dennard scaling pushes for hardware parallelism instead of faster single-threaded execution; finally, virtualization hardware in commodity servers has become commonplace providing means to multiplex this cheap parallel hardware.

7 Dissertation Map

The dissertation proceeds as follows. Chapter 2 provides background on our prior research, which motivated this work. Section 2.2 discusses related work and research context. Chapter 3 introduces Dynamic Privilege, our OS model, the kElevate system call and how we propose to marry the application and kernel worlds. Chapter 5

discusses the supporting mechanisms we developed to facilitate the use of Dynamic Privilege in a general-purpose OS, Linux. Chapter 4 provides a set of demonstrations of Dynamic Privilege. Section 4.2 explores the use of Dynamic Privilege to optimize applications. Chapter 6 concludes the dissertation and discusses future work.

Chapter 2

Research Context

This chapter proceeds in two major sections. The first section describes my prior work that informed the creation of Dynamic Privilege. Because this work was done in collaboration with multiple other researchers, I will tend to use "our work," unless specifying individual contribution. The second major section highlights related work from the wider space of academic discourse and industrial efforts, mainly in the area of OS architecture. Figure 2·1 illustrates and organizes the related work covered in this chapter.

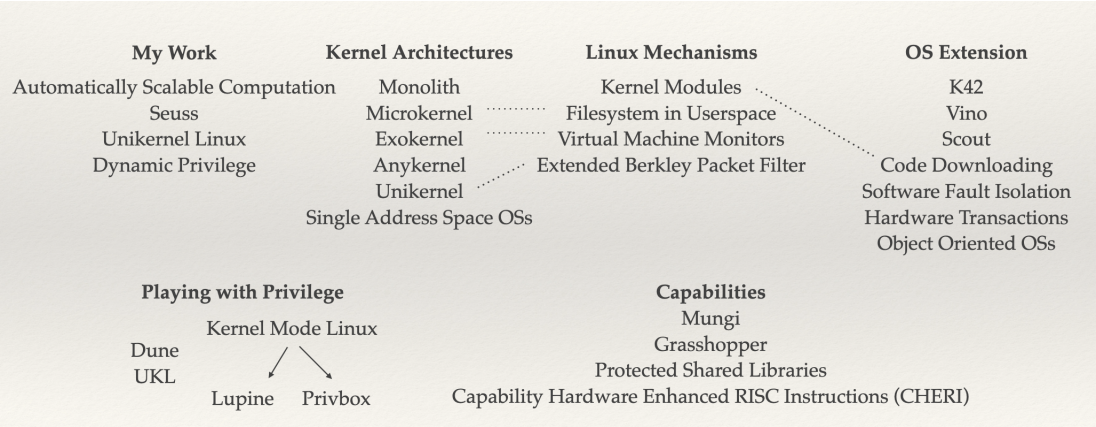

Figure 2·1: The first half of this chapter addresses my prior academic work, including the four listed systems. The second half of this chapter discusses various kernel architectures and existing mechanisms implemented by the Linux OS. We draw parallels between these mechanisms and the academic discourse.

Dynamic Privilege developed from insights and experiences gained during our work to enhance application performance in three specific scenarios: 1) Building a user-level runtime monitor that transparently accelerated a Linux process. 2) Creating

a custom OS-level monitor capable of enabling transparent acceleration of library OSs containing only unprivileged code. 3) The development of application-specific optimizations that take advantage of the absence of runtime privilege boundaries in a uni-kernelized version of Linux.

One major use case for Dynamic Privilege is improving the "flexibility" of systems. Informally, flexibility is the ability to change the core properties of an operating system, such as extending application-OS interfaces, specializing, and reimplementating system internals. The second section of this chapter Section 2.2 considers alternative OS architectures such as Monoliths, Microkernels, Exokernels, and Unikernels. These architectures occupy very different regions of the system design space, and achieve different trade-offs in terms of application performance, security, and resource use. We also consider various Linux mechanisms such as code downloading, kernel modules, upcalls, vDSO, eBPF, Ftrace, and Kprobes which allow some degree of flexibility to explore the local area around the Linux design point. Ultimately, we believe Dynamic Privilege brings a new kind of flexibility to modify systems. This exploration of alternative architectures demonstrates some extreme points in design space and the corresponding trade-offs that a flexible system may be able to navigate between at runtime.

1 Background and Our Prior Research

This section overviews our prior work, consisting of three software systems used to improve application performance. We discuss Automatically Scalable Computation (ASC), Serverless Execution with Unikernel SnapShots (SEUSS), and Unikernel Linux (UKL). Each system performed different performance optimizations, detailed in the corresponding papers (Waterland et al., 2014; Cadden et al., 2020; Raza et al., 2023).

To structure our overview in terms that will be useful for our later discussion

of Dynamic Privilege, we highlight three concepts from OS architecture,[1] 1) hardware privilege, 2) virtual address spaces, and 3) application-kernel interfaces (see Section 2.1.1). Further, we introduce a framework called the "Source-to-Runtime Spectrum" to highlight critical periods for optimization in the lifetime of software construction and execution. This spectrum comprises source code, executable formats, and runtime execution (see Section 2.1.1).

In Section 2.1.2, we examine ASC, SUESS, and UKL using the concepts and terms developed in Section 2.1.1.

[1]By "system architecture," we refer to the overall components and interactions within both the OS and applications.

Background: System Architecture Concepts

Privilege, Virtual Memory, Application-Kernel Interface

Source-to-Runtime Spectrum

Source Code

Executable Formats:

- Executable Linker Format (ELF)
- A "ready" or "runnable" kernel/user thread
- State Vector (SV) or Snapshot

Software at Runtime:

- A "running" kernel/user thread or event handler

Prior Work: Studies in Performance Optimization

ASC: Computation as data object

runtime → SV → transformation → SV → runtime

SEUSS: adding a user privilege sandbox to a Unikernel

runtime → snapshot → runtime. Anticipatory Optimization

UKL: adding an elevated process to a GPOS

custom interfaces via. source code editing

Figure 2·2: Section Organization.

1.1 Background: Privilege, Virtual Memory & Interfaces

Operating system architecture describes a computer system's overall design principles and software structure. Architecture involves the organization and interaction of OS components such as the kernel, device drivers, system calls, and application-kernel interfaces. This organization significantly influences fundamental system properties such as performance (Chen and Bershad, 1993; Patterson and Hennessy, 1990; Dahlin et al., 1994), fault tolerance (Liedtke, 1996; Gray, 1988; Chapin et al., 1995), security

(Madhavapeddy et al., 2013; Saltzer and Schroeder, 1975), and scalability (Feeley et al., 1995; Lamport, 1998). Different operating systems implement various structures to cater to specific needs and scenarios, resulting in diverse designs with unique trade-offs as we discuss in Section 2.2 Section 2.2.2. This dissertation concentrates on a subset of trade-offs, specifically focusing on hardware privilege, virtual memory, and application-kernel interfaces.

While we defer the description of our prior research systems to Section 2.1.2, we include Table 2.1 here to motivate our discussion of hardware privilege, virtual memory, and application-kernel interfaces.

Architectural Comparison of Thesis Systems

System→ Property↓	**ASC**	**SEUSS**	**UKL**	**Dynamic Privilege**
Goal	Parallelize sequential processes automatically and transparently.	Optimize FaaS platform transparently.	Enable Unikernel optimizations of a Linux process.	Enable system evolution and optimization from app. software ecosystem.
Hardware Privilege	All work was done with user privileged processes.	Extend supervisor mode only EbbRT Unikernel with user mode Fn. sandboxes.	Extend the Linux kernel with an optimized process that runs with supervisor privilege.	Enable threads to toggle privilege status independently.
Virtual Address Space	A user process monitor captures a target process' VAS as a data-object, operations on it, and reinstantiates it.	An OS-level monitor snapshots FaaS Unikernels, caching them to amortize deterministic initialization costs.	An app. binary is statically linked into the kernel, and runs in the kernel's portion of the VAS.	When running with privilege, the kernel portion of the VAS becomes accessible to app. process.
Application Kernel Interface	Relied heavily on existing kernel interfaces designed to allow debuggers to inspect and control another target process.	The Snapshot system call requests the OS-level monitor to capture the Unikernel.	The optimized process gains access to the kernel symbol table (enabling it to call standalone kernel functions and access kernel data).	Add kElevate System Call. With additional tools, gains access to kernel symbol table, and additionally all the type, macro, and inline functions of the kernel.

Table 2.1: This table compares my work on the three systems that influenced the development of Dynamic Privilege to Dynamic Privilege itself. It highlights their relations to three architectural elements: hardware privilege, virtual address space, and the application-kernel interface.

Hardware Privilege

General-purpose OS kernel software primarily uses hardware privilege mechanisms to separate applications from the kernel.[2] The mechanisms are used to: 1) restrict access to system instructions and associated system registers, and 2) control and virtualize the memory and devices of the system. Given this structure, the kernel offers a set of functions known as system calls, which applications can invoke through specific unprivileged instruction sequences to access kernel-provided services.

General-purpose processors typically offer two (or multi) level privilege separation to isolate a trusted kernel from untrusted applications; applications are restricted to a subset of memory access and the Instruction Set Architecture (ISA). One key observation about hardware privilege is that the privileged mode of execution (typically called the supervisor mode) can perform a proper superset of the functionality available to the unprivileged mode.

As an example, most hardware systems feature "general purpose" registers and instructions, enabling application software to perform arithmetic and logic operations in both unprivileged and privileged modes. Typically, these systems also include "special purpose" or "system" registers and instructions, which only execute successfully in the processor's privileged mode. For instance, loading the page table register, crucial for virtual to physical address translation,[3] is an example of such privileged operations. Our notion of Dynamic Privilege builds directly on the fact that, although application binaries today only use unprivileged instructions, there is no fundamental restriction in the hardware against mixing privileged and unprivileged instructions together (as done in kernel code).

[2]Uncommon exceptions exist such as password and capability-based systems(Heiser and Elphinstone, 2016; Woodruff et al., 2014). Specialized embedded systems also may not support virtual memory.

[3]Page tables and the address translation they control are fundamental for kernel software to isolate applications from each other.

Virtual Address Spaces

Operating systems typically organize code and data in one or more linear virtual address spaces. Unikernels, a form of special purpose OS (see Section 2.1.2 and Section 2.2.3) utilize a single address space for both the application and the OS. In contrast, Microkernels (see Section 2.2.3) minimize the privileged kernel functionality, and usually separate the OS and application functionality into distinct address spaces. Modern Monoliths like the UNIX and Berkeley Software Distribution (BSD) families of OSs execute applications in opposite ends of the same address space. When context-switching, these systems trade out the user portion of the address space.

Modern processors couple virtual memory addressing and memory access control in a structure called the Memory Management Unit (MMU).[4] Page tables program MMUs via translation data structures like x86 or ARM page tables. When using paging (standard practice for most OSs), these tables encode mappings from virtual pages to physical memory frames. They also contain a host of privilege bits determining if and now a virtual memory page can be accessed (executed, read, and written). Most undergraduate system textbooks, such as Bryant and O'Halloron(Bryant and O'Hallaron, 2016), provide an overview of typical MMU functionality and paging.

Application-Kernel Interfaces

Operating systems vary in the level of abstraction of the primitives they present to applications. We briefly contrast the Monolithic, Microkernel, and Exokernel approaches here; for more detail, Section 2.2.2. Monolithic operating systems, for instance, typically offer high-level logical primitives that generally do not correspond directly to hardware resources, such as files and processes. The internal implementation and interfaces to these primitives are hidden and immutable to users. This

[4]Exceptions to this rule include past use of tagging and segmentation and capability systems past and present.

approach allows applications to interact with complex resources using simple, well-defined interfaces (albeit potentially large in number). In contrast, Microkernel-based operating systems tend to provide only a small set of more basic, hardware-agnostic primitives, such as address spaces, threads, ports, and interprocess communication (IPC). These primitives are the building blocks applications use to implement more complex systems, like file systems, as application-level servers.

Exokernels, in contrast to Microkernels, advocate for abstractions that directly reflect common hardware mechanisms, supporting direct bindings to hardware resources. The goal is to grant applications greater control, permitting them to construct abstractions that directly suit their needs. In Exokernels, functionalities typically managed by the operating system, such as file systems and network protocols, are implemented in user-space libraries, which will explicitly become part of the application and under its control. This approach, distinct from Monolithic and Microkernel architectures, shifts the responsibility for creating complex abstractions to the application, aiming for maximum flexibility and potential performance gains.

System designers balance many issues when deciding where to implement functionality with respect to the privilege divide. They balance user customization, with practical and consistent interfaces, control and arbitration, sharing, and security. System models vary from centralized and fully privileged implementations, such as a file system integrated into a Monolith, to the decentralized, distributed, and heterogeneous mixes of implementation characteristics of both Microkernels and Exokernels. This spectrum of designs reflects a trade-off between centralized control and the flexibility of decentralized systems.

The Source-to-Runtime Spectrum

This subsection introduces the "Source-to-Runtime Spectrum" to discuss the range of significant stages in the software lifetime and allows us to observe how the stages

and boundaries between them impact optimization. While different OS architectures prescribe the placement of system functionality on either side of the hardware privilege divide, our most crucial point is that:

> OS architectures that separate applications from kernels (Monoliths, Microkernels, and Exokernels) maintain a strict separation between the two entities at all stages of the Source-to-Runtime Spectrum.

Consider Figure 2·3, which illustrates the extent of this separation: isolated source codebases make independent trips through compilers producing distinct binaries, which execute in hardware-isolated ends of an address space or separate address spaces altogether.

We use the term "Source-to-Runtime Spectrum" to describe the key stages in software's lifetime: source code, executable formats, and runtime software execution.[5] Each stage presents different optimization opportunities and possesses distinct knowledge about the program.[6] Dynamic Privilege intersects this spectrum at the end, runtime. Yet its repercussions influence system design all the way back to the source-code level, as elaborated in Chapter 4.

To understand the Source-to-Runtime Spectrum, it may help to refer to Figure 2·3. OS architectural choices determine what functionality to implement in the kernel or user code. Binary executables are files from which a running program or application can be initiated. Each operating system utilizes a specific file format, which is the

[5]This is a coarse-grained overview of the numerous translation steps from source code to executing programs, including preprocessing, compilation, assembling, linking, etc.

[6]Consider the familiar case of an optimizing compiler and linker. The compile applies language-specific front-end optimizations to the source code of a compilation unit, including constant folding. Dead code elimination can be performed later when the intermediate representation is available. Later still, the compiler back-end can perform architecture-specific register allocation optimization. Just before producing an executable, link-time optimization may perform cross-module inlining. Finally, using profile-guided optimization, the compiler instruments an executable to collect statistics at runtime, exploited in a final recompilation to optimize branch layout, for example(Contributors, 2023).

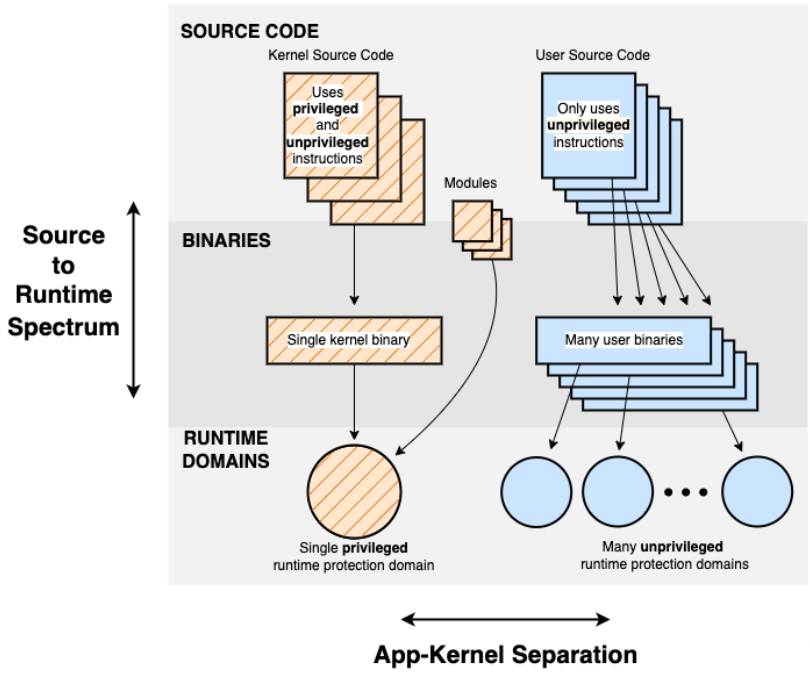

Figure 2·3: We refer to this model as the Source-to-Runtime Spectrum. We use it to analyze the application-kernel separation throughout the software lifecycle. Segregated OS and application source codebases compile into separate executables (binaries). On UNIX-like systems, these exist as separate ELF files. These binaries are loaded and booted as separate runtime domains. These runtime domains, namely the kernel and user runtime domains, are executed in secluded corners of the same address space or separate address spaces entirely, depending on the OS. At runtime on Linux, the Memory Management Unit (MMU) uses privilege to separate applications and the kernel at runtime.

method of encoding file information in bytes, enabling the operating system kernel to launch a new running instance (a process on Linux) from it. These files are commonly referred to as "executables" or "binaries." Binary Executable File Formats — In LINUX, binary executables adhere to a file format known as "Executable and Linkable Format (ELF)."[7] As illustrated, a single kernel binary and zero or more application binaries are produced (with the possibility of inserting kernel modules at runtime). The kernel runs with privilege while the user programs do not. **Notice that the kernel and user entities are kept separate at all stages.** We use these various stages of the Source-to-Runtime Spectrum to organize our presentation of our prior research and observe their impact on the optimizations we explored. See the Table 2.2.

Isolating the application and kernel at all stages along the Source-to-Runtime Spectrum eases security concerns during implementation (see Section 3.1.3). While security is a major concern for general-purpose OSs, this separation precludes many opportunities for optimization across the kernel-application divide, including compilation[8] and specialization.[9] For this reason, there was particular interest in finding controlled ways to "extend" these systems (see Section 2.2.2). Through this lens, it is little surprise that some OS developers pursued the Unikernel or libraryOS model where application and kernel code intermingle at some or all stages in the Source-to-Runtime Spectrum. Unikernels offer a greater opportunity for specialization and optimization but eschew functionality such as multiprocessing, thus dedicating work-

[7] On OSX and iOS, executables use the "Mach-O" file format. For Windows, "Portable Executable (PE)" is a prevalent format. Another example is the "eXtended Common Object File Format (XCOFF)," used by IBM. Our discussion and examples will primarily focus on Linux and ELF files. In our research, we will refer to alternative executable formats such as State Vectors and Snapshot Stacks (see Section 2.1.2 and Section 2.1.2).

[8] The Mirage Unikernel (Madhavapeddy et al., 2013) does not divide its software in this way, and as a result, it can find and eliminate dead-code more effectively.

[9] The EbbRT framework for building per-application Unikernels allows applications to specialize the system stack such as installing application-specific hardware event handlers.

loads to physical or virtual hardware. We dive deeper into the Unikernel model in Section 2.2.3.

System→ Property↓	**ASC**	**SEUSS**	**UKL**	**Dynam. Priv.**
Source	N/A	N/A	Adding new kernel interfaces for the optimized process. Modify glib to shortcut all app. system calls to corresponding kernel handlers.	Utilize kernel module buildsystem to use all kernel types, macros and inline functions to include kernel code into elevated process.
Executable Format	State Vector: a full representation of a process at an instant in time. Operating on the SV and reinstatiating it as a new process.	Snapshot: a full representation of a function in the Unikernel context. Snapshot stacks offer massive memory sharing.	Optimized app. binary statically linked to kernel ELF. Opportunity for compiler optimization.	Completely standard application ELFs contain elevated code.
Runtime	User-mode monitor samples target process, produces SVs, and verifies speculative computation.	OS-Level runtime monitor produces and deploys snapshot stacks. Anticipatory Optimization: speculatively execute dummy workloads ahead of snapshotting to pre-warm app. and kernel code paths.	Single optimized process co-runs with potentially many user-level processes.	Executing potentially many elevated (and standard) processes. Option to transparently optimize unmodified apps.

Table 2.2: Source-to-Runtime Spectrum: Different points along the source-to-runtime spectrum offer different opportunities for optimization intervention we have explored in various research systems and models. We consider three course-grained stages during the software lifetimes of these systems. These are the source code, the executable format, and runtime execution.

1.2 Our Prior Research

We discuss three systems from our prior work: ASC, SEUSS, and UKL; see Section 2.1.2. We used each system to optimize application performance transparently (no application source modification) or, optionally, with limited source modification. ASC demonstrated the capability of capturing runtime execution as a data object, operating on that object, and then redeploying it. In SEUSS, we added user privilege sandboxes to the EbbRT Unikernel and brought snapshot caching to a Functions as a Service (FaaS) platform. Finally, in UKL, we extended Linux to allow a single process to run alongside the kernel, with privilege, enriching the process system call interface to include all internal functions of the kernel. Each system performed different performance optimizations, detailed in the corresponding papers (Waterland et al., 2014; Cadden et al., 2020; Raza et al., 2023).

ASC: Computation as Data

It would be an understatement to say that developers find it easier to write sequential software than performant parallel software. ASC attempts to parallelize the execution of programs automatically, so programmers don't have to. Our prior work on Automatically Scalable Computation (ASC) explored the transparent optimization of sequential processes. ASC explored capturing the entire state of a process into a data object and transforming that object using approximate computing, sometimes on heterogeneous hardware, before reconstituting the process for execution by a CPU. We provide an overview of the project, then discuss the lessons we bring to our later work on SEUSS and Dynamic Privilege.

Overview: ASC is a transparent runtime monitor (see Figure 2·4) that aims to speed up the wall clock runtime of sequential processes by executing subcomponents of the program in parallel, on other cores (Waterland et al., 2014; Eldridge et al.,

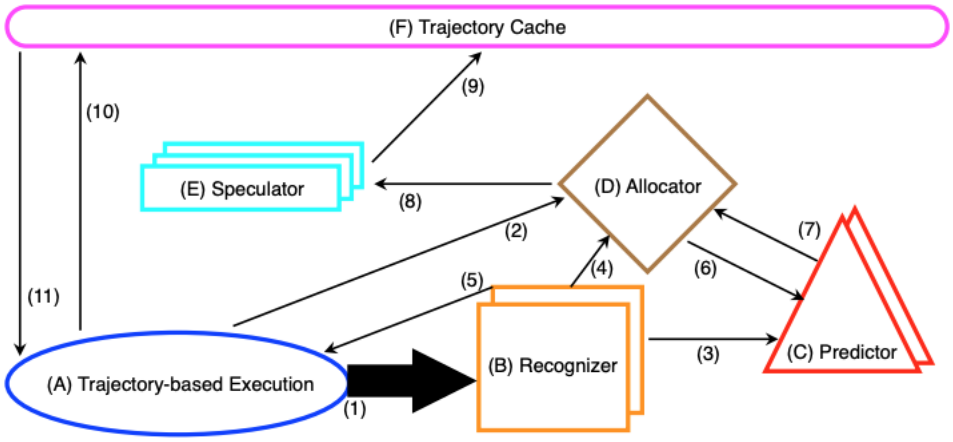

Figure 2·4: Figure 1. ASC Architecture. The trajectory-based execution engine forwards its current state to the recognizers (1) and allocator (2). Recognizers identify key states for the predictors, allocator, and execution engine (3, 4, 5). Predictors then model future states from the recognizers' filtered states. The allocator, which assigns components to cores, utilizes these states to guide predictors in generating future states (6, 7) and to dispatch speculative threads with varying predicted states and instruction counts (8). These threads update the trajectory cache with their start-state/end-state pairs (9). Periodically, the execution engine checks the cache for matching start-states (10), adopting the farthest end-state for continued operation (11).

2015a; Perez et al., 2016). ASC operates according to a speculative execution model. While ASC is a software monitor, the concept of speculative execution may be more familiar to the reader through the example of hardware branch predictors.[10] Unlike hardware branch prediction, which operates over dozens of cycles, ASC functions on a much larger timescale, spanning tens of millions of cycles. In the best case, ASC provides linear speedup, dividing a process's sequential runtime by the number of available cores. Analogous to a poorly performing branch predictor, in the worst case, ASC adds its overhead to the sequential execution time. While it is possible for ASC to slow down the execution of a target by introducing overhead, we emphasize

[10]Branch predictors, integral to processors, optimize performance by addressing hazards conditional branches introduce to the instruction pipeline. These branches create uncertainty in instruction flow, which branch predictors manage by "guessing" the next direction, guided by statistical heuristics from past execution. Speculative execution leverages this to enhance microarchitectural parallelism and overlaps I/O activities, such as memory fetches. Branch prediction typically improves software performance as measured by instructions-per-cycle but can degrade performance if it predicts poorly.

that ASC *cannot* commit to erroneous computation.

In ASC, a monitoring process samples the complete state of the "target process," the sequential program we want to speed up, as it executes. Statistical algorithms, such as neural networks, compare a series of past and present process state pairs, attempting to approximate a "transition function" that maps the previous state to the current state on a bit-wise granularity (literally predicting which bits in the virtual address space and registers will change and what their values will be). If the predictor's error rates drop below a threshold, ASC applies the transition function instead to the *current* state to synthesize a speculative future state.[11] ASC instantiates these speculations by constructing new parallel processes on other cores, buffering output so these potentially incorrect trajectories don't emit erroneous results to the outside world. If execution of the (ground truth) target proves a speculation correct, ASC accelerates computation to whatever state that speculative core reached, emits its I/O, and destroys the other candidates.

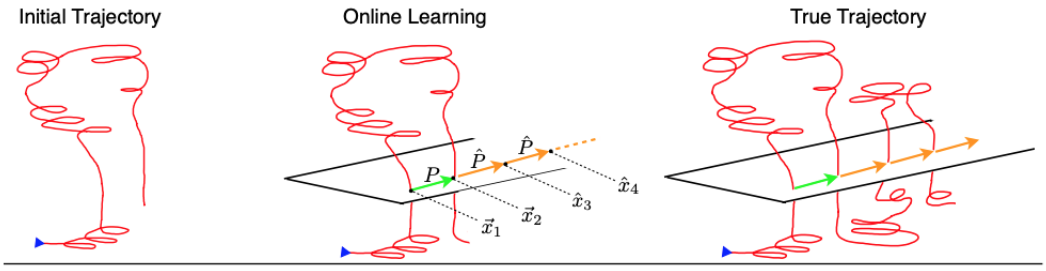

Figure 2·5: From the initial trajectory, we recognize a hyperplane that cuts the true trajectory at regular, widely-spaced points $\{\vec{x}_1, \vec{x}_2, \ldots\}$, all sharing the same IP value. Our online learning problem is then to learn an approximation $\hat{P}$ to the true function P that maps $\vec{x}_i$ to $\vec{x}_{i+1}$. Once we have learned $\hat{P}$, we make predictions as $\hat{x}_{i+1} = \hat{P}(\vec{x}_i)$.

ASC exhibited substantial, often nearly linear speed-up in both simulated and real-world environments. The system mainly operated on various computational "kernels," mainly mathematical or graph traversal programs, as illustrated in Figure 2·6 (see an

[11] ASC can trade-off between breadth-first (brute-forcing the bits it is most uncertain of) and depth-first (recursive prediction) approaches depending on the difficulty of the prediction problem.

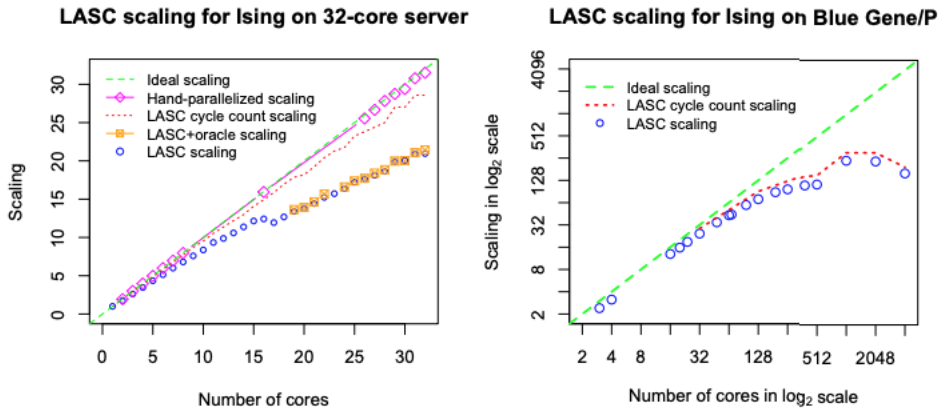

Figure 2·6: Simulated scaling results for Ising benchmark.

"Ising" energy minimization example).[12] However, this approach has its limitations. One major challenge is determining where and when to sample the target process' state. Formalized as "hyperplanes" (see Figure 2·5), these sampling criteria critically affected ASC's ability to predict future states and accelerate the target.

"Good" hyperplanes are determined through a combination of heuristics such as reducing the number of "excited bits" (bits seen to vary between samples), as well as minimizing the "entropy," or difficulty of predicting those excited bits. One interesting finding is that prediction difficulty is *not* simply proportional to prediction "distance," or how far forward in the instruction stream the prediction is made.[13] Finally, ASC is sensitive to both the underlying predictability of the program and the structure of the input.

Dealing with OS state maintained outside the process virtual address space (files,

[12] An excerpt from the ASC paper (Waterland et al., 2014), "Our first benchmark is the Ising kernel, a pointer-based condensed matter physics program. It came to our attention because our colleagues in applied physics found that existing parallelizing compilers were unable to parallelize it. The program walks a linked list of spin configurations, looking for the element in the list producing the lowest energy state. Computing the energy for each configuration is computationally intensive. Programs that use dynamic data structures are notoriously difficult to automatically parallelize because of the difficulties of alias analysis in pointer-based code. We demonstrate that by predicting the addresses of linked list elements, LASC parallelizes this kernel."

[13] A simple example that demonstrates this is a for loop that hashes the increment variable, then zeros it. Predicting the result of the hash is hard, but it becomes easy when the register is cleared.

sockets, network connections, etc.) is complicated. ASC largely restricted itself to applications fully described by their internal virtual address space contents. Computation within ASC's target process can be optimized, but there is no opportunity to optimize the host OS or anything else outside the target process. See our next project, SEUSS, which generalizes this by capturing almost all such state within its optimization domain (see Section 2.1.2).

My work on ASC focused on the speculative state prediction problem. 1) I explored using alternative neural network architectures, replacing multi-layer perceptron networks(Rojas, 1996) running on CPUs with much higher throughput GPU-accelerated convolutional networks. We used these convolutional networks to identify semantic structures, such as integer counters from process traces, though we did not incorporate this into the runtime system. 2) When ASC encountered new portions of the virtual address space (such as during dynamic memory allocation), its fixed-width neural networks were discarded, and larger ones were retrained from scratch. I worked on two solutions to this problem, one for more flexible software networks, and another for static hardware networks (such as FPGA networks). First, I engineered the Fast Artificial Neural Network (FANN) library to support "expanding" networks, allowing us to add new input and output nodes to an existing network, preserving prior training while adding new computational capacity. Second, I implemented Bloom filter fingerprinting to map arbitrarily large input vectors to fixed-width input and output vectors. Both of these approaches improved target application performance because of improved prediction rates. 3) ASC performance is limited by its overhead. One way to reduce neural network execution time is to accelerate it using parallel hardware like Field-Programmable Gate Arrays (FPGAs). I assisted in debugging an FPGA-based neural network accelerator used to optimize the latency of neural network computation, including ASC (Eldridge et al., 2015b).

Architectural Lessons: ASC provided us with experience capturing running process state efficiently. ASC demonstrates the optimization potential in treating the Source-to-Runtime Spectrum of Section 2.1.1 as bi-directional; moving computation from a runtime phenomenon back to executable formats is both possible and useful for optimization.

ASC is not part of the OS; it operates at the user-process level. ASC performs a transformation on a running process, which gathers the complete contents of a target process's virtual address space into a static data object. We call this data object a "State Vector" (SV). There is an inverse operation that scatters an SV back out into a process that is ready for execution. This SV is a (non-running) executable representation (as discussed in the Source-to-Runtime Spectrum in Section 2.1.1) of the target process. Applying this technique at a larger scale formed the backbone of the "snapshotting" method we utilized in our next project, SEUSS (see the following Section).

Operating as a user process, ASC leveraged complex algorithms and libraries, a task that would have been challenging to implement within an OS kernel or a VM monitor. However, its functions align closely with typical OS tasks, and we suspect (guided from experience on SEUSS) that direct access to the OS data structures for process state representation could have significantly reduced overheads. Producing State Vectors from running processes (and the inverse) required deep copies, a major cost SEUSS had the opportunity to eliminate because it had access to the hardware page table representation. Providing an efficient method for user processes to access these OS-level data structures would have entailed considerable effort, likely outweighing the benefits. Ideally, ASC would have benefitted from selective access to kernel-level data structures and code, utilizing them as needed without a full-scale integration of a new application-kernel interface.

SEUSS: Smaller, Faster Functions as a Service

The Function-as-a-Service (FaaS) model of computation aims to provide clients with simple access to cloud resources in a high-level pay-as-you-go format. Clients upload high-level source code scripts (in languages such as Javascript and Python), which cloud providers execute on demand, potentially with high degrees of parallelism. Clients do not maintain servers (as they would when renting virtual machines), and providers retain control of the entire software stack that executes these functions, allowing them significant latitude to optimize on the back end.

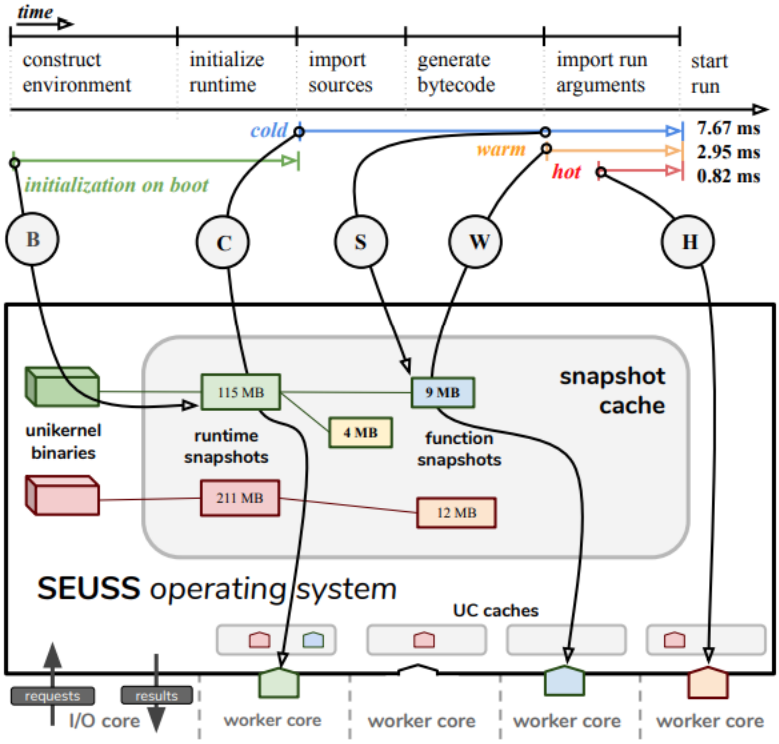

Figure 2·7: The high-level operations of a SEUSS OS deployed on a FaaS compute node.

FaaS providers need to create backend execution environments for functions. They must instantiate these execution environments efficiently and ensure sufficient isolation between potentially distrusting workloads. The "cold-start" problem refers to the significant time cost associated with starting a function that has either not been ex-

ecuted previously or not been used recently. This extended startup time can notably limit the utility of small, infrequent, or dynamically generated functions. The goal for SEUSS(Cadden et al., 2020) was to develop a drop-in replacement for the backend of a FaaS system, improving performance, memory usage, and isolation properties over the containerized baseline, OpenWhisk (Contributors, 2016a). Our approach relied on computational caching (see Figure 2·7) in the form of page table-based snapshotting and deployment. This enabled 1) the elimination of latency from amortized deterministic boot processes; 2) massive memory sharing, much more than even (poorly isolated) processes could achieve; 3) Anticipatory Optimization: this speculative approach involves pre-warming critical code paths–an intuitive approach to pre-warming the initial use of network and bytecode compiler paths. Given SEUSS's more efficient representation, it could cache approximately 16 times more functions than the standard container-based approach. This improvement in density provides the FaaS platform with the new ability to handle large-scale bursts of requests.

The key enabling step for SEUSS was shifting away from instantiating functions as processes, containers, or VMs as other platforms used, instead encapsulating both the application and OS together as a Unikernel. When the application and OS run as a single address space, a monitor layer can "snapshot" the Unikernel by copying the page table and the underlying data it maps.[14] This snapshotting method produced a data object conceptually similar to ASC's SVs (see Section 2.1.2). High-level language interpreters can be hundreds of megabytes, so naive snapshotting is impractical, as fewer than a thousand snapshots would fit in memory on a server with around a hundred gigabytes of memory. SEUSS utilized "snapshot stacks" which allowed for taking function-specific snapshots as memory "deltas" that were applied to a shared

[14]The Unikernel approach also offers benefits to function isolation. We interfaced these Unikernels to our host OS through the narrow Solo5(sol, 2021) interface. While applications inside the Unikernel enjoy the full set of hundreds of system calls, the Unikernel interfaces to the host OS through only 12 hypercalls, dramatically reducing the attack surface.

"base snapshot," enabling the storage of around 16,000 unique function snapshots (on a node with 128G of memory). When using snapshot stacks, it is preferable to deduplicate memory into the base snapshot, or as close to it as possible (in the case of many snapshots). We developed an intuitive approach to realizing this called "Anticipatory Optimization" (AO). With AO, we doubled our function cache size to 32,000 and dramatically reduced cold start function initialization times by a factor of 5.6x. We describe AO next.

Anticipatory Optimization (AO) profoundly improved our system's performance, particularly impacting function memory footprint and initialization times. By proactively warming up the function software's internal pathways and data structures, including both application and OS components in the Unikernel stack, AO effectively prepares the system before capturing the base runtime snapshot. This is achieved by sending dummy network traffic and compiling a dummy script. As a result, executions spawned from these snapshots circumvent time-consuming allocation and initialization procedures, leading to faster startup and execution times. Furthermore, the technique improves the cacheability of function-specific snapshots by reducing the number of written pages captured in each snapshot. With AO, the memory footprint of the *single* base snapshot grows (by 4.9Mb), but all of the tens of thousands of function-specific snapshots shrink from 4.8Mb to 2.0Mb. Cold start functions initialization times fell from 42ms before AO, 16.8ms when warming just the network stack, and finally to 7.5ms when also warming the bytecode compiler using AO. Warm start times similarly fell from 7.6ms to 3.5ms.

Putting all of SEUSS's optimizations together, Faas platform throughput improved by 51x (see Figure 2·9) on workloads composed entirely of new functions, stressing the cold-start paths. We managed to cache over 50,000 function instances in memory instead of 3,000 using standard OS techniques. In combination, these

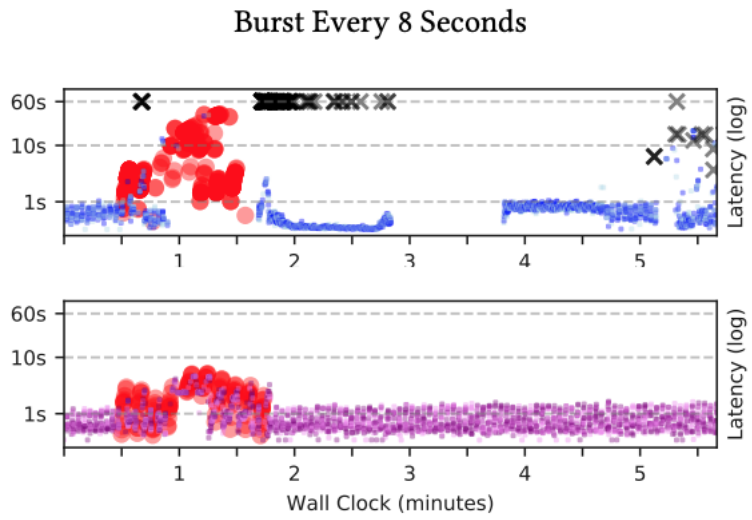

Figure 2·8: FaaS platforms Top: Linux, Bottom: SEUSS. Each system serves a background rate of function re-executions. Starting at the 30s mark, bursts of cold-start functions are requested. The Linux backend is unable to keep up with the bursts and also drops the background traffic. SEUSS keeps up with the burst and maintains the background traffic. Linux also failed to support bursts sent at 16s and 32s intervals.

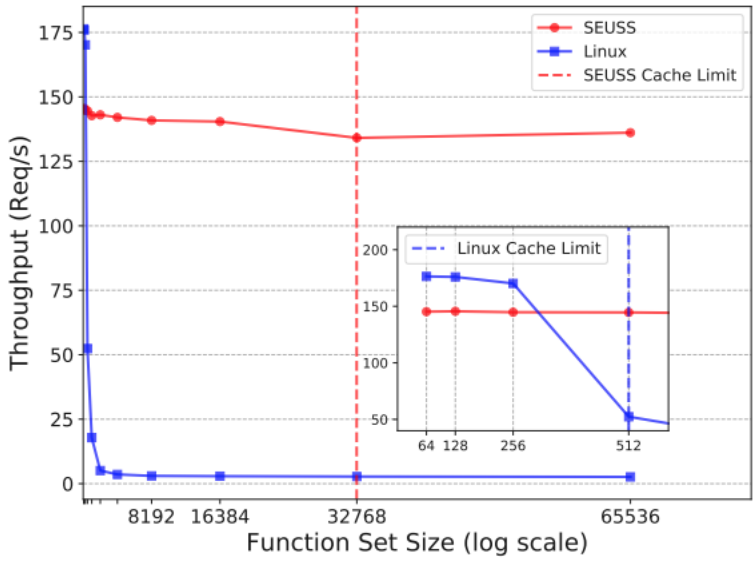

Figure 2·9: OpenWhisk Platform Throughput. While Linux outperforms SEUSS throughput at the smallest function set sizes, SEUSS scales more gracefully with little fall-off from warm to cold-start heavy workloads.

improvements give the FaaS platform a new ability to handle large-scale bursts of requests (see Figure 2·8). Notice the semi-log axis used for both graphs.

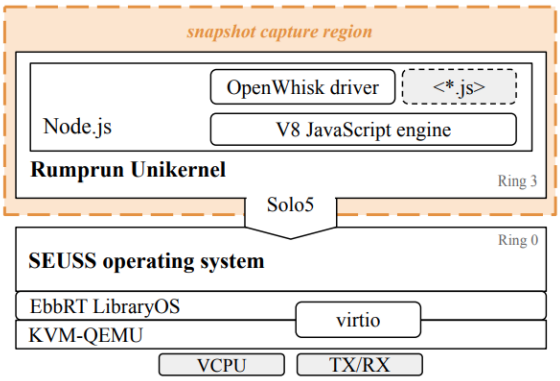

Figure 2·10: Single-core vertical slice of the software stack of our SEUSS prototype.

SEUSS acted as a drop-in replacement for the OpenWisk FaaS platform. SEUSS is written in C++ and extends the EbbRT libraryOS framework, totaling 7,971 lines of new code. EbbRT provides the bottom-most software layer. On top of EbbRT, we implemented the functionality to deploy and multiplex Unikernel contexts, capture snapshots, and route function network traffic to the external network. SEUSS runs in protection ring 0 (kernel mode), while function instances execute in protection ring 3 (user mode). Internal to every Unikernel context is the Rumprun Unikernel linked with a port of Node.js or Python. This version of Rumprun was previously ported to run in userspace on top of Solo5 middleware library, which defines a very small set of domain-crossing hypercalls, reducing porting effort. We used the Rumprun Unikernel (Kantee, 2012), which offered a significant degree of compatibility in its support for multiple high-level language interpreters like Javascript and Python. EbbRT provides the majority of low-level functionality that SEUSS requires (e.g., a multicore event-driven run-time, a virtio paravirtualized NIC, and a zero-copy TCP/IP network stack). Rumprun provides a POSIX-like Unikernel based on the NetBSD source with a common set of shared system libraries and a ramdisk filesystem.

My contributions focused on the back-end function invoker. I extended EbbRT with user-privilege process-like sandboxes. This is where high-level language interpreter Unikernels ran in their virtual address space and at the user-privilege level. I developed an ELF loader to load these Unikernels into memory. Most of my work focused on developing the recursive page table snapshot mechanism. As you can see in Figure 2·10, we run our function Unikernels on top of the solo5 interface tender. I extended the Unikernel with a snapshot system call which would trigger a hypercall through the Solo5 layer, requesting EbbRT to snapshot the environment. Within EbbRT, I developed page table walking algorithms responsible for duplicating the translation structures, the underlying data they map, and the general purpose register state. For the implementation of snapshot stacks, these tools selectively make deep copies of *dirty* pages only, recognizing that pages that have not been written are already stored in the system's memory, and will be shared copy-on-write. Lastly, I discovered our use of anticipatory optimization and explored its application on SEUSS.

Architectural Lessons ASC and SEUSS capture runtime computation as executable data objects, though they do this differently. ASC was a user-space monitor producing State Vectors through software interfaces. By contrast, SEUSS operates at the OS level, creating snapshots by walking hardware page table structures. Because SEUSS captured Unikernel environments, it could see across the application/kernel interface. This was important in the use of anticipatory optimization which applied across the software stack, warming both kernel and interpreter internals. AO requires running the computational target past the naive point required for replay to load the cached data object with state speculatively.

SEUSS involved the addition of a user-space sandbox to the EbbRT OS, which had only previously operated with privilege in supervisor mode. These process-like

sandboxes held Unikernels in independent address spaces, facilitating sandboxing, multiplexing, isolation, and independent failure domains.

ASC benefitted from using a number of high-level application libraries, but likely suffered overhead from the lack of access to low-level hardware. We suffered in the opposite way in SEUSS where page table operations were often fast (measured in microseconds), but we had to implement so much low-level functionality ourselves and were restricted to using software supported by EbbRT to implement our optimizations. This project may have been easier on Linux if we had access to Dynamic Privilege. We might have written the main SEUSS OS functionality as user-level processes, dipping into supervisor mode for the lowest-level operations, and reusing Linux kernel code snippets whenever possible.

UKL: Specializing a General-Purpose OS

Unikernels have proven their ability to optimize application performance, but they are often so specialized that they lack basic "compatibility" with standard software ecosystems. In particular, they often require manual application porting, including significant source code rewriting. They often only run a single process, eschewing supporting tools like debuggers, logging services, etc. Unikernels often only run on a single core, and only in virtualization, supporting simple virtualized device interfaces.

The Unikernel Linux project, UKL(Raza et al., 2023), aims to address this by extending Linux to run a single Unikernel-optimized process alongside any number of standard user-level processes. Instead of building a highly optimized kernel, UKL's "base model" runs with little to no immediate performance improvement. From the base model, however, we demonstrate that it is possible to optimize the process incrementally. Standard Linux processes interface with the kernel via the system call interface. UKL statically links the Unikernel optimized process into Linux kernel, opening up the wide internal kernel interface, which offers opportunities for opti-

mization. UKL was designed with upstreaming as a main goal. If upstreamed, it will enjoy ongoing maintenance from the community and may lead to architected support for further Unikernel optimizations.

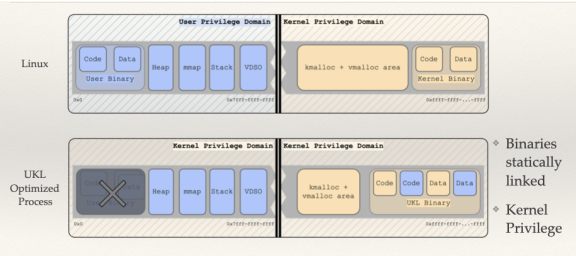

Figure 2·11: Comparing the UKL Optimized process to a standard Linux process.

UKL's optimized process runs multithreaded and multicore. It runs in virtualization and on baremetal deployments using Linux's device drivers. The optimized binary (comprised of the code and data sections of the process) is statically linked into the Linux kernel, with the dynamic sections (like the heap, mmap, region, stacks, and VDSO) allocated in the lower half, like a standard process (see Figure 2·11). The application threads always run with supervisor privilege. The UKL-optimized process becomes part of the trusted compute base, like the kernel itself. Thus, UKL is assumed to operate as part of the trusted compute base on dedicated virtual or physical resources.

To demonstrate the value of running a process as a UKL-optimized Unikernel, we developed an optimization technique called "shortcutting" (see Figure 2·12). Because the optimized processes are statically linked into the Linux kernel, they can make function calls into kernel paths. Making use of a modified glibc. At the base level, this flattens syscalls into function calls. Further, it allows multiple degrees of optimization: selectively invoking system entry and exit code, as well as providing "deep shortcutting" where modified applications use their access to Linux's internal interfaces as a library, skipping general-purpose paths of expensive marshaling. UKL

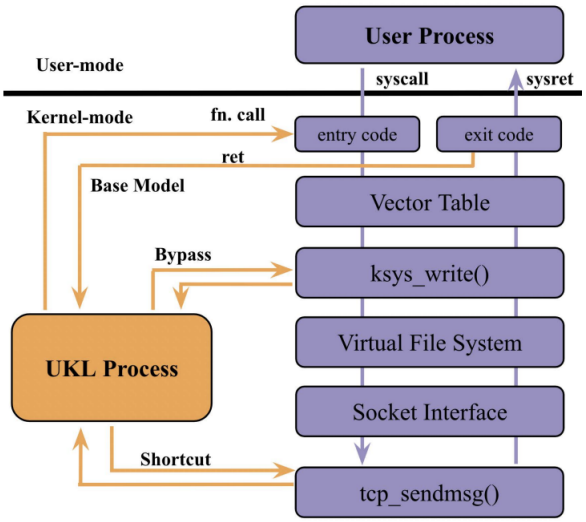

Figure 2·12: A schematic of a write system call destined for a network device. The three alternative internal entry points that a UKL process exercises are shown in orange.

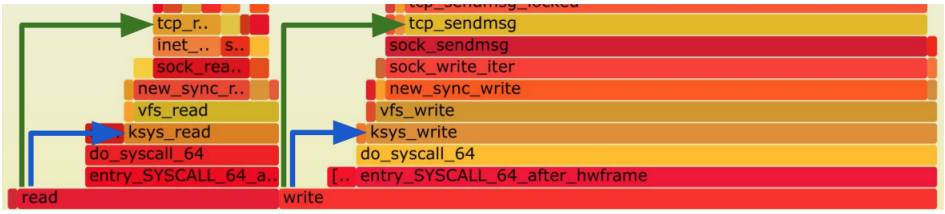

Figure 2·13: Part of a flame graph generated after profiling Redis-UKL base model with perf. The read and write functions at the bottom reside in Redis code. Blue arrows show the code bypassed in UKL_BYP, and green arrows show deeper shortcuts.

does not provide ABI compatibility as it requires a recompilation when building the app to the kernel model, but it does not require source code modification for its base set of optimizations.

UKL introduces a new "transition" model to execute a user process within the kernel space. Traditionally, transitions between the kernel and a user process are implicitly indicated through hardware events associated with a change in privilege level (e.g., system call instructions, interrupts, and exceptions). During these transitions, the operating system's "transition code" is executed to 1) set up the execution environment correctly for either the kernel or user process, depending on the transition

type, and 2) ensure that kernel bookkeeping and periodic code, such as RCU clocks, are executed. However, the UKL process does not undergo privilege-level transitions. As a result, new mechanisms must be implemented to track transitions and execute existing and/or new transition code, ensuring that the user or kernel execution environments are appropriately set up. For example, UKL introduces new code and stack segment selectors for the UKL process. In addition to the transition model described above, we have also made minor modifications to the Linux kernel. These modifications include modifications to page fault handling to avoid the "Stack Starvation" issue (see Section 5.3.2).

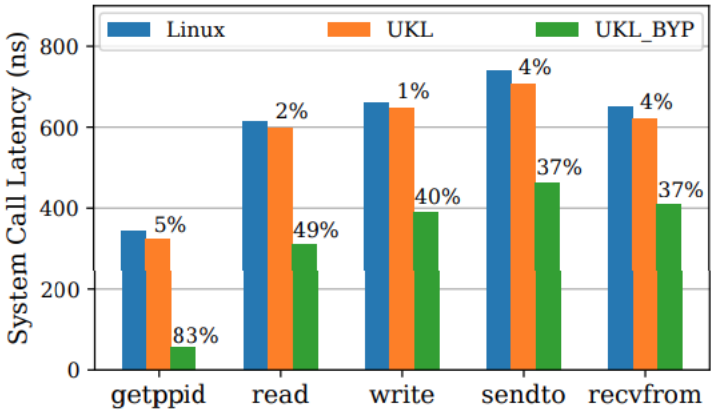

Figure 2·14: Comparison of Linux, UKL base model, and UKL with bypass configuration for simple system calls. With modern hardware, the UKL advantage of avoiding the system call overhead is modest. However, there appears to be a significant advantage for simple calls with UKL_BYP to avoid transition checks between application and kernel code.

We evaluated UKL on multiple microbenchmark (see Figure 2·14) and macrobenchmark (see Figure 2·15 and Figure 2·16) suites. When running the Redis in memory key-value store, UKL achieved throughput improvements from shallow shortcutting of 12%, and 26% for deep shortcuts. 99% tail latency improvements of 11% and 22% respectively.

At UKL's inception, the system was designed to run one privileged process without

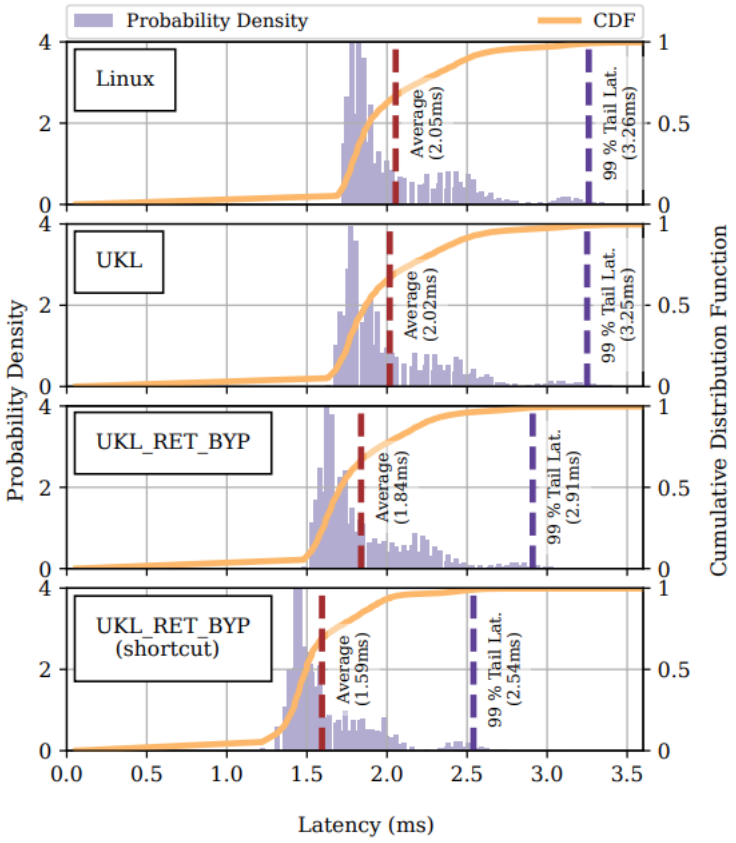

Figure 2·15: . Probability Density (purple bars) and CDF (orange line) of Redis deployed on Linux, UKL, UKL_RET_BYP and UKL_RET_BYP (shortcut) and tested with the memtier_benchmark. Average latency (broken red line) and 99th percentile tail latency (broken purple line).

	99 % tail lat		Throughput	
System	**(ms)**	**Improv.**	**(Kb/s)**	**Improv.**
Linux	3.26	-	6375.20	-
UKL base model	3.25	0.3%	6479.20	1.6%
UKL_RET_BYP	2.91	11%	7154.68	12%
UKL_RET_BYP (shortcut)	2.54	22%	8022.54	26%

Figure 2·16: Redis throughput and latency improvements of UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut) over Linux

any co-running user-space processes. A significant design contribution I made was to modify this limitation, allowing multiple user-level processes to coexist with the

privileged process. The desire to do this was driven by the challenges of using the `perf` subsystem to profile the UKL process, which is crucial for understanding the impact of our optimizations on system performance. My contributions also included debugging the system call and interrupt transition code for the privileged process, and extending the kernel linker script to support full standard process (and C library) bring-up and tear-down. This included providing constructor and destructor support, which made running C++ programs in addition to C possible. Additionally, I developed and tested UKL's deep shortcutting technique, which leverages UKL's specialization to invoke custom kernel entry points, optimizing the system's overall functionality.

Architecture Lessons UKL was our first experience executing standard Linux applications with kernel privileges. UKL employed a static linking approach, mapping the Unikernel-optimized application binary to execute in the kernel half of the virtual address space and always running with hardware privilege, akin to a conventional Unikernel. UKL Unikernels are capable of independently executing kernel functions and accessing kernel data. The project significantly expanded the application-kernel interface, by making it possible to call internal kernel routines from application code.

While UKL retains significant compatibility with Linux and takes strides towards opening up the internal kernel interface for application specialization, it falls short of providing the rich application execution environment familiar to developers, which we list here:

1. Iterative development on a UKL-optimized application is cumbersome, necessitating the relinking of the application library into the Linux kernel: adding a print statement to the program implies rebooting the node.

2. UKL uses static linking of the application to the kernel, and changing application build systems (along with all their dependent libraries) to build statically

is a non-trivial engineering task.

3. UKL supports only one optimized process at a time, restricting workloads.

4. UKL, like other Unikernels, does not support the standard `fork` system call.

5. Every thread in a UKL process runs with elevated privileges throughout its entire lifecycle. This impacts iterative development because "nothing works until everything works."

6. UKL does not support using prevalent orchestration tools, such as containerization, thereby limiting its practicality in modern development environments.

1.3 Conclusion

In conclusion, our prior research on ASC, SEUSS, and UKL informed our work on Dynamic Privilege. ASC demonstrated the potential for optimizing runtime execution by capturing and transforming computational state, while SEUSS showed the benefits of combining application and OS together in a Unikernel to enable efficient snapshotting and caching. UKL extended the Linux kernel to support running a single Unikernel-optimized process alongside standard user-level processes, offering opportunities for incremental optimization through an expanded application-kernel interface.

However, the limitations encountered in these projects, particularly in UKL, have motivated us to explore the Dynamic Privilege model as a means to address the challenges associated with static linking, iterative development, and the lack of support for standard development tools. Building on this experience, we aim to enable a more flexible and developer-friendly execution environment that retains the performance benefits of Unikernels while providing a rich application execution environment familiar to developers.

2 Related Work

In the first section of the chapter, we considered the work that directly influenced and led to the development of this dissertation's core thesis, Dynamic Privilege. We now move on to discussing privilege in the broader systems literature. Operating system research has long focused on the static separation of privileged and unprivileged system components. Extensive research has considered how a system's structure influences its ability to optimize application performance and affect resource consumption. This section offers a review of academic OS models (or architectures) as well as industrial OS mechanisms that interact with the notion of where code is placed concerning the privilege boundary. While we believe Dynamic Privilege (see Chapter 3) offers a more flexible approach to exploring these tradeoffs with the benefit of high-level tooling at runtime, these tradeoffs can still be informed by prior work in static settings.

2.1 OS Architectures and Privilege

Over the years, four major OS architectures have arisen: Monoliths, Microkernels, Exokernels, and Unikernels. These four architectures represent a significant portion of OS diversity and, thus, a significant portion of the known trade-off space (elaborated below). These models differ in their selection of which software will run with hardware privilege. Monoliths, Microkernels, and Exokernels explicitly separate the application and kernel codebases at every stage, while Unikernels combine the two into a single binary, running them in the same virtual address space and hardware privilege level. Despite these differences, the majority of general-purpose OSs implement some form of "code downloading," enabling the injection of new code into the kernel at runtime. While not the same as dynamically allowing a domain to change its privilege, the ability to add functionality to a privileged domain provides a degree of dynamic

freedom.

The differences between these architectures are not superficial; they determine core properties, such as the functionality and level of abstraction of application-kernel interfaces. Architectural choices directly impact critical system trade-offs, including the types and extent of performance optimization, fault tolerance, and the kernel's suitability for formal verification. The following section discusses these architectures with respect to research systems that have been constructed to explore them. After presenting the research systems, we also address mechanisms within modern general-purpose Monoliths like Linux, which provide features similar to those found in alternative research kernels. These include Kernel Modules, upcalls, Virtual Machine Monitors (hypervisors), and Linux's vDSO and eBPF mechanisms, acknowledging the existing degree of runtime flexibility they provide. The final section expands our discussion to include an overview of existing operating system archetypes, using Linux as a case study to illustrate mechanisms found in modern general-purpose operating systems and research systems.

Research Systems Research projects often explore novel approaches to specific challenges, such as improving performance or security, while OS architectures provide a broader framework for system design. While most of this section focuses on architecture, here, we mention some specific projects as exemplars of research areas seen in Figure 2·1. One notable research project is Kernel Mode Linux (KML)(Maeda and Yonezawa, 2003), which allowed Linux processes to run with privilege and employed language verification techniques to prevent tight coupling between these privileged applications and the OS. This approach aimed to enhance performance by reducing the overhead associated with system calls. Privbox(Kuznetsov and Morrison, 2022a), built on KML, created a privileged compartment within a process, similarly a reduction in the hardware overhead of system calls. These projects demonstrate the

potential for using privilege to gain performance optimization. In contrast, capability-based systems like Cheri(Woodruff et al., 2014) take a different approach by decoupling access from privilege, offering a fine-grained hybrid software-hardware model for byte-granularity memory safety.

2.2 Binary Organization and Code Downloading

Before jumping into a comparison of various kernel architectures and mechanisms, we highlight "binary organization" (described next) and code downloading as two concepts that help organize our study in this section. Binary organization is a concept that helps us describe where software functionality is placed with respect to the privilege divide. Code downloading is an approach to extending running kernels by many research and commodity OSs. Code downloading offers a restricted approach to modifying a system's binary organization at runtime, demonstrating a demand for system flexibility.

Binary Organization: What Software Should Run with Privilege?

Section 2.1.1 introduced the Source-to-Runtime Spectrum, a model for analyzing the application-kernel separation throughout the software lifecycle. In the following sections, we focus on the concept of "binary organization" to examine how various OS architectures implement functionality. Binary organization encompasses two main aspects: 1) the allocation of functionality between application and kernel binaries, and 2) the placement of these binaries within the virtual address space at runtime. The design decisions surrounding binary organization significantly impact system performance, security, and flexibility, making it a central concern of OS architecture. For example, traditional monolithic operating systems place most OS functionality within the kernel, running in a privileged mode with full hardware access. While this approach may reduce context-switching overheads, it also results in a larger failure

domain.

Alternative OS architectures, such as Exokernels, Microkernels, and Unikernels, challenge the traditional Monolithic binary organization wisdom. Exokernels provide applications with low-level hardware access through library operating systems (libraryOSs), enabling fine-grained optimization and customization. Microkernels implement high-level system policies as user-space servers, improving fault isolation and modularity, potentially at the cost of increased inter-process communication overhead. Unikernels take an extreme approach, linking application and OS functionality into a single binary, allowing for aggressive co-optimization and a minimized attack surface. By exploring different points in the OS functionality placement design space, these architectures demonstrate the inherent performance, security, and flexibility trade-offs in the privilege boundary decision. This inspires building flexible systems that can easily navigete this trade-off space.

Code Downloading: Breaking the Mold

These systems offer varying degrees of flexibility when it comes to modifying the "structure" of the system. Here, structure means the choices of where functionality is placed with respect to system-enforced boundaries. Modifying certain system abstractions, like replacing a Mach(Accetta et al., 1986) system server running in userspace, modifying an Exokernel system library before compiling it with an application, or a Unikernel exploiting an internal kernel path are notable examples. Other system models offload performance-critical portions to virtual environments with direct access to virtual hardware. This is seen in systems like Dune and EbbRT, where traditional kernel abstractions can be decomposed such that applications can manage their hardware representations (Dune(Belay et al., 2012), EbbRT(Schatzberg et al., 2016)).

Code downloading is a well-studied area in computer systems. Code downloading

techniques span static configuration (Mosberer and Peterson, 1996), object-oriented kernel extension(Krieger et al., 2006; Small and Seltzer, 1994), to software fault isolation(Engler et al., 1995b), verification(Bershad et al., 1995; McCanne and Jacobson, 1993), and transactions(Small and Seltzer, 1994). We found this overview from the perspective of OS extension very useful (Seltzer et al., 1997), as well as the discussion in (Kaashoek et al., 1997). Linux Kernel modules are a special case of code downloading (Kernel-Contributers, 2022). We will revisit code downloading in the context of Linux Modules in more detail in Section 2.2.5.

Code downloading is of particular interest to this dissertation because it is a mechanism by which the binary organization of the kernels gets extended with code that runs with kernel privilege. One use case of code downloading is to move application functionality into a system protection domain. As such, it can be used to achieve performance optimizations such as specializing in the kernel interface or implementing custom code paths.

2.3 Classic OS architectures

The academic world presents system architectures that significantly deviate from today's Monolithic general-purpose OSs. In this section, we discuss Exokernel(Engler et al., 1995b), Microkernel(Accetta et al., 1986), and Unikernel(Madhavapeddy et al., 2013) system architectures and their derivatives. These are models that stake out unique points in the binary organization design space, far from the commodity Monoliths the reader may be most familiar with. These systems exemplify how binary organization is often the defining characteristic of a system's design. In particular, we will consider how these architectures demand a rethinking of traditional application design.

Exokernel

> The Exokernel architecture is founded on and motivated by a single, simple, and old observation: the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementers of higher-level abstractions.
>
> – Exokernel paper (Engler et al., 1995a)

The Exokernel challenges the conventional–Monolithic–approach of imposing unchangeable system abstractions on developers, aiming instead to separate resource protection from system management. Ideally, the Exokernel exports raw hardware, and the application, or "library operating system" (libraryOS), constructs all abstractions on top of that. This is to alleviate the observed problem that, "These abstractions define a virtual machine on which applications execute; their implementation cannot be replaced or modified by untrusted applications" (Engler et al., 1995b).

Architecturally, the Exokernel comprises a minimal kernel, multiplexing possibly distrusting library OSs (libraryOSs), running at user-level and with interrupts enabled. Raw physical hardware is exposed to these libraryOSs. It may seem surprising that it is possible to offer physical resources to untrusted libraryOSs. The trick involves separating "authorization" from "access" using caching. Kernel authorization is required for cache element creation but *not* for access. Adding a technique to invalidate the cache (the abort protocol) allows the Exokernel to break a libraryOS's bindings and, thus, access to hardware. A simple example is software TLB entries, but this same approach is used for more complicated mechanisms like downloading typesafe code into the kernel.

In terms of binary organization, the Exokernel departs from the Monolithic model by lifting abstractions like processes, files, address spaces, and inter-process communication out of the kernel binary, placing them in system libraries available for libraryOS

developers to modify and link application code with. The key to optimization in the Exokernel is enabling libraryOSs low-level access to hardware. Contrast this with issuing a system call on a Monolithic operating system: the binary organization of the libraryOS enables the compiler to see the app and OS code at the same time, whereas an app running on a Monolith issues syscalls into a black box. The libraryOS has the opportunity to choose which library will implement its components as well as the ability to modify or replace them. The libraryOS can implement its own bespoke policies and abstractions over the hardware to match its requirements.

The Exokernel implements code downloading. Within the Exokernel, untrusted "Application Specific Safe Handlers" may be inserted and guarded by code inspection and software sandboxing. As the kernel guarantees the safety of this downloaded code, the mechanism can be used by untrusted applications. Code is downloaded for two named reasons. First, to reduce context-switching costs. Second, because downloaded code can be "tamed" or bounded in execution time, it can be run when the application is not scheduled, even when there are only a few microseconds of idle time. Packet filters are an example of this.

An interesting iteration of this work came seventeen years later in 2012, with Dune(Belay et al., 2012). Dune has a very similar aim, providing safe high-performance application libraryOSs low-level hardware access. Dune employs virtual machines (contrasting Exokernel's secure bindings) for isolation and exposes a process interface to virtual environments. Dune detracts from the idea of exterminating all OS abstractions by exposing a process interface (not the typical machine interface) to these virtual environments: "Dune provides access to privileged hardware features so that they can be used in concert with the OS instead of a means of modifying or overriding it." This is the critical difference: working with the grain of general-purpose abstractions when possible, then only overriding them when they constrain

optimization.

Microkernel

> "The actual system running on any particular machine is a function of its servers rather than its kernel."
>
> "... allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel."
>
> – Mach paper (Accetta et al., 1986)

Microkernels like Mach (1986) implement high-level system policies in userspace as servers instead of within the kernel, improving fault isolation but incurring inter-process communication (IPC) costs. Microkernels enable users to modify or replace system servers running in userspace. They decouple system policy in terms of location and protection. System policy is largely liberated from the location (and control) of kernel address space.

System policies can be modular and implemented in languages or programming models that do not conform to the kernel's model, with multiple implementations running concurrently and failing independently. The Microkernel model also separates system policy from hardware privilege. Instead of implementing all system policies in kernel mode, only policies that need protection from user modification or require access to privileged instructions need to run within the kernel, while the rest are deployed in userspace (where a segmentation fault does not imply a kernel panic).

In terms of binary organization, Microkernels lie somewhere between the Exokernel and Monolithic design. Microkernels eject the filesystems, schedulers, and memory managers from the kernel binary but retain IPC mechanisms and virtual memory management.

Mach's architecture supports the task, the thread, the port, and the message as key abstractions. A task is a collection of system resources, including a virtual address space and a set of port rights (capabilities). Threads are the basic computation units, with multiple threads per task: "the Mach abstractions of task, thread, and port correspond to the physical realization of many multiprocessors as nodes with shared memory, one or more processors and external communication ports." Mach IPC is built using ports and messages. Ports are protected kernel objects that hold messages, and access to a port is granted by receiving a message containing a port capability. Mach contains many developments in the areas of virtual memory and IPC, but these are considered out of the scope of this conversation.

Advances in hardware, such as fast syscalls, tagged translation lookaside buffers (TLBs), IPC mechanisms, and large core counts, have encouraged rethinking of Microkernel approaches (Marty et al., 2019) (Humphries et al., 2021). These days, Microkernel approaches are motivated by a range of issues, from kernel verification(Heiser and Elphinstone, 2016) to low latency scheduling, networking, storage, and continuous integration.

Unikernel

> We take an extreme position on specialization, treating the final VM image as a single-purpose appliance rather than a general-purpose system by stripping away functionality at compile-time.
>
> – Unikernels: Library Operating Systems for the Cloud (Madhavapeddy et al., 2013)

The term "Unikernel" was introduced by Madhavapeddy et al. in 2013 (Madhavapeddy et al., 2013) as a specific type of library OS. It was initially intended to denote a single-purpose, highly optimized, application-specific libraryOS, designed to

run on virtual hardware. Optimization goals included improving performance, security, and boot times through compiler specialization. However, over time, "Unikernel" has elevated in abstraction to become a synonym for "library OS." In line with its current usage within the community, we will use this term to refer to the general case throughout this dissertation.

Unikernels closely associates application code with traditional system code. In terms of binary organization, Unikernels occupy an extreme point in the design space, with all application and system code linked into a single binary. Depending on the system, this binary may be hosted by various platforms, including virtual machines (MirageOS), processes(Unikernels as Processes), and even self-hosting on baremetal hardware(UKL, Rumprun) (Madhavapeddy et al., 2013) (Williams et al., 2018) (Raza et al., 2023) (Kantee, 2012).

Unikernels typically enable co-optimization across the application-OS boundary. Application and kernel code often go through the same compilation or linking step and run in the same protection ring. This allows for aggressive specialization, like dead code elimination(Madhavapeddy et al., 2013), as well as compile-time or link-time optimization(Raza et al., 2023). It also permits the application to fully utilize the wide libraryOS internal interface, as opposed to the narrow syscall interface presented by a Monolith. Linking application code directly with system code allows for a fine-grained approach to optimization, something that is not possible with Monolithic or Microkernel approaches.

2.4 Linux and its Mechanisms

Having discussed the major architecturally distinct OS models from the research world, we turn to the state-of-the-art general-purpose OS design, Linux. As a Monolith, Linux consists of a large kernel binary that implements a wide range of high-level system primitives and policies. We have discussed the benefits of alternative models

in the context of research OSs, and Linux has incorporated many of these lessons while retaining its Monolithic design. In this section, we will discuss the Linux kernel and its mechanisms like kernel modules, upcalls, vDSO, eBPF, Ftrace, and Kprobes, drawing connections between these mechanisms and the OS architectures we described in the previous section.

The kernel module subsystem (Kernel-Contributers, 2022) serves as the backbone of Linux's approach to runtime kernel modification via code downloading. Interestingly, the kernel module system does not attempt to maintain "compatibility" through this mechanism. At a coarse grain, "compatibility" here is the notion that software such as kernel retains its essential properties; for example, the Linux kernel does not leak information about the virtual address space locations of its data structures. Yet, a kernel module may violate that property (breaking compatibility) by printing the address of a data structure to dmesg. While the research systems we discussed use techniques like type safety, verification, and sandboxing to ensure that downloaded code does not violate system invariants, the base model mechanisms of Linux does not attempt to do so. Instead, compatibility is handled orthogonally (if at all) through community code review, automated testing, and software verification. Decoupling compatibility from extension simplifies implementers' efforts in creating simple and powerful tools. This degree of freedom keeps the module mechanism simple.

VMMs as Exokernel Analogs

The Exokernel approach has an analog in Xen (Barham et al., 2003) and the Linux Kernel Virtual Machine (KVM) (Contributors, 2019), which monitors and multiplexes virtual machines. Both Exokernels and VMMs systems allow untrusted guest operating systems and applications and maintain simple low-level interfaces with the host OS or Exokernel. In KVM's case, this is the machine-level Hypercall Interface. Within virtual environments, guests have access to virtual hardware. However,

instead of Exokernel's secure bindings to actual hardware, KVM may run in emulation or use virtualization extensions, such as x86's Extended Page Tables and virtual IOMMU, given supporting CPUs and motherboards.

Performance differences exist between using raw hardware and virtual extensions. Both models provide cut-throughs to address the performance cost of separating the kernel from the guest environment: Exokernel offers safe code downloading, while KVM supports various paravirtualization techniques like memory ballooning, paravirtualized disks and network devices. IX (Belay et al., 2014) compares the two: "Similar to the Exokernel, each dataplane runs a single application in a single address space. However, we use modern virtualization hardware to provide three-way isolation between the control plane, the dataplane, and untrusted user code."

Upcalls as Microkernel Analogs

Monolithic kernels, in contrast to Microkernels, implement system services within the kernel, typically to reduce context-switching overheads or to provide reuse through a convenient interface to high-level abstractions via syscalls. One Linux kernel mechanism that resonates with the Microkernel approach is upcalls, where the kernel invokes a function in userspace without a slow call to `schedule()`. This mechanism could be used to implement scheduler activations(Anderson et al., 1991) that initiate application threads when the kernel hits a delay, such as blocking I/O. The FUSE (Contributors, 2020) project uses upcalls to implement filesystems in userspace, allowing for a hybrid between Monolithic and Microkernel approaches. From their documentation:(Contributors, 2020) "FUSE (Filesystem in Userspace) is an interface for userspace programs to export a filesystem to the Linux kernel." User applications still issue open(), write(), etc. syscalls through the standard kernel interface (this is the Monolithic part), while the FUSE kernel module issues upcalls into a userspace filesystem (the Microkernel part).

vDSO and eBPF as Unikernel Analogs

Unikernels focus on the library operating system (application) of an Exokernel design. Two mechanisms in Linux mirror the system-application co-optimization characteristic of Unikernels: vDSO, which places kernel code in userspace, and the (Extended) Berkley Packet Filter (Contributors, 2016b) which places user-specified code in kernel space. From the docs(Linux man-pages project, 2021), "The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications." This enables functions like `gettimeofday()` to avoid boundary crossings associated with syscalls and to offload architecture-dependent particularities of accessing the timestamp counter. In contrast, eBPF allows system and application developers with administrator credentials to write scripts in userspace that can be injected into the kernel if they pass a static analysis conducted by an in-kernel verifier. eBPF is interesting from an extension perspective, both in its implementation and use. We will return to it in the next section.

2.5 Code Downloading in Linux: Kernel Modules, eBPF, and Friends

In the previous section, we discussed mechanisms in the Linux kernel that mirror architectural approaches from the academic literature. This section delves deeper into Linux kernel modules, which serve as the backbone of Linux's approach to runtime kernel modification. We examine Ftrace (Contributors, 2008) and Kprobes (Contributors, 2006) to develop a detailed understanding of how these systems enable the insertion of modules. We conclude with the eBPF system, which uses static analysis to constrain the powerful kernel module subsystem. eBPF attempts to wrangle the unrestricted kernel module system into a less powerful but safer, and simpler form.

Kernel Modules

The kernel module subsystem attempts to address the modularity and extensibility limitations of Monolithic kernels. It allows for runtime kernel modification and serves as a foundation for other extensions. Kernel modules package device drivers, reducing the kernel's memory footprint and attack surface to only the loaded modules. Linux distributions use modules to package and distribute software differences from the upstream kernel.

Implemented with an in-kernel dynamic loader, kernel modules allow arbitrary code to be inserted into the kernel (with sudo access). Code must include kernel headers and not link with user libraries. Further, it must be built in agreement with the kernel model and turned into object code. These position-independent shared objects are read, memory is allocated, and they are loaded into the kernel. Finally, a symbol resolution pass is done, and they are inserted. There is a reverse process for removal. Inserted modules typically run as event-driven handlers.

The kernel module system is powerful because it allows arbitrary code to be inserted into the kernel (with sudo access). The base mechanism does not protect the kernel and other processes from errant or malicious modules, but maintainers strive to accept good code and address bugs iteratively.

Module Hooks: Ftrace and Kprobes

Ftrace(Contributors, 2008) and Kprobes(Contributors, 2006) place hooks into the kernel, allowing handler calls to be placed using the module system. Ftrace is partially automated, with the compiler inserting a five-byte nop instruction at the beginning of every function that can be binary rewritten at runtime. Ftrace is partially hand-built, with the kernel source instrumented with tracepoints at semantically interesting locations like critical scheduling branches. Ftrace and kernel modules facilitate debugging

through printing and logging tools.

Kprobes builds on and extends the Ftrace infrastructure, generalizing the locations where handlers can be inserted with few limitations. This is achieved by copying kernel instructions to a buffer and overwriting that instruction with an `int3` instruction, generating a synchronous breakpoint exception (on `x86`). The copied instruction is executed in single-step mode, and control is handed off to a handler inserted as a module, as with Ftrace.

Extended Berkeley Packet Filter

The extended Berkeley Packet Filter(Contributors, 2016b) (eBPF) aims to defang the kernel module system, allowing safe code injection on kernel paths; that is, it attempts to constrain the kernel module approach so an intervention is less likely to compromise the system. Application and system developers write scripts to be analyzed by an in-kernel verifier and then installed using the Kprobes system. The verifier statically verifies scripts, attempting to accept only safe ones. eBPF enables system administrators to customize kernel paths, such as performance-sensitive networking paths for network filtering, debugging, prototyping, and application acceleration. eBPF maps facilitate data sharing between user and kernel spaces.

A schism within the eBPF community revolves around whether unprivileged users should be allowed to install an eBPF handler. The crux of this issue lies in the verifier's ability to reliably determine script safety. The Linux kernel community's stance is that, no, only users with sudo credentials can use the system. While eBPF scripts execute in supervisor mode within the kernel, they specifically prohibit access to the privileged instruction set. From a security perspective, it is clear that unprivileged users should not have the ability to overwrite core data structures like page tables. However, upon further reflection, an interesting observation emerges: eBPF scripts run in supervisor mode within the kernel not because they require access to the

extended instruction set but rather to access kernel data on kernel code paths and make it available to user applications.

3 Conclusion

The first section of this chapter addressed our prior research on ASC, SEUSS, and UKL. Our experience building these systems provided insights into the relationships between OS architecture, the Source-to-Runtime Spectrum, and opportunities for application performance optimization. These systems demonstrated alternative approaches to capturing, transforming, and redeploying computation, as well as the potential benefits of blurring the traditional boundaries between applications and the kernel. The lessons learned from these projects, particularly regarding hardware privilege, virtual memory, and application-kernel interfaces, prepared us for the development of Dynamic Privilege. We identified the need for a more flexible and powerful approach to optimizing application performance without sacrificing compatibility with existing software ecosystems.

The second section of this chapter surveyed a wide range of operating system architectures, highlighting their approaches to managing the privilege divide between the kernel and user applications. These models provide distinct perspectives on how to navigate the trade-off space such as performance and resource use in system design. Furthermore, we examined how modern general-purpose operating systems incorporate mechanisms like kernel modules, upcalls, vDSO, eBPF, Ftrace, and Kprobes to introduce runtime flexibility and extend kernel functionality. Despite the many years and range of designs, we find that in the end, they all utilize a static divide with respect to privilege.

This chapter has laid the foundation for understanding various approaches to managing the kernel-user privilege boundary and the implications of these approaches

on system design. The next chapter will discuss the Dynamic Privilege model and a simple mechanism that implements it on Linux. This new tool connects application-level tools to low-level system access. It promises an easier route to engaging in the types of trade-offs explored in this chapter.

Chapter 3

Dynamic Privilege, The kElevate Mechanism & OS-Level Access

Before presenting the details of this chapter, we briefly summarize the core of the dissertation as context. This dissertation navigates beyond the traditional boundaries of OS software organization, challenging the conventional wisdom that programs should operate strictly with or without hardware privilege. We introduce and explore Dynamic Privilege, an OS model enabling credentialed threads to independently toggle their hardware privilege (e.g. between user and supervisor modes on x86 hardware) for code sequences of any length (executing for an arbitrary amount of time). The Dynamic Privilege OS model separates hardware privilege access as an independent consideration from other OS functionality. It provides elevated threads access to the extended Instruction Set Architecture (ISA), allowing them to use system registers and execute all instructions. Additionally, it liberates threads from the memory access constraints programmed by translation data structures (like page tables) and enforced by the Memory Management Unit (MMU). When applied in the context of a general-purpose OS, Dynamic Privilege extends the limited system call interface, enabling elevated threads to interact with kernel data and to modify and execute kernel code paths.

This chapter comprises four sections. Section 3.1 introduces the Dynamic Privilege model, which decouples hardware privilege from control flow transfer. Section 3.2

discusses the kElevate mechanism, a system call we add to the Linux kernel to enable exploration of Dynamic Privilege in the context of a general-purpose OS. Chapter 4 introduces our efforts to marry application and kernel codebases, exposing kernel data and code paths to elevated processes; a primary practical result of this work is to provide elevated processes the ability to link with a shared library that exposes kernel symbols dynamically. Finally, Section 3.3 provides a three-part framework for understanding the novel impact of implementing the Dynamic Privilege OS model in the context of a general-purpose OS, bringing OS-level **access** to the application domain.

1 Dynamic Privilege as an OS Model

We begin with an upfront discussion of the Dynamic Privilege OS model. Because Dynamic Privilege operates close to the hardware level, it is relatively simple to describe. The implications of this foundational change, however, are wide-ranging. The second subsection introduces our use of the concept of "access" (explored in greater detail in the final section of this chapter), describing why we enact our intervention at the hardware privilege level. The third subsection explores the traditional coupling of hardware privilege with control flow transfer and the use of Dynamic Privilege to decouple this.

1.1 Dynamic Privilege

> **Dynamic Privilege:** *The ability of a unit of execution, such as a thread, to independently transition between different hardware execution modes.*

Expanding on the Definition

The independent transfer of hardware modes of execution characterizes Dynamic Privilege. Specifically, these transfers are independent from the typical control flow

transfer mechanisms of the OS, such as system calls, interrupts, and exceptions. Dynamic Privilege is a model an OSs may realize, not a specific implementation. It does not make architectural assumptions about the OS or hardware, requiring only the hardware's capability for transitioning between privilege levels. The design of Dynamic Privilege aims for implementation via a low-level mechanism.[1] Unlike conventional OS models, Dynamic Privilege enables a process thread loaded from a single binary to run sequences of code alternating between execution with and without hardware privilege. This approach contrasts with traditional OS models, where a simultaneous toggle of hardware privilege occurs with a control flow shift from user to kernel runtime,[2] or vice versa.

Atributes

The Dynamic Privilege Model: Key Atributes

- **Hardware-Centric:** The definition is grounded in computational hardware, facilitating integration with existing hardware privilege mechanisms.
- **OS-Agnostic:** It makes no assumptions about the OS architecture, meaning it applies to various types such as Monolithic, Microkernel, Exokernel, or Unikernel systems.
- **Thread Privilege Flexibility:** The model does not dictate which threads can switch their privilege or their initial privilege state, e.g., kernel or user threads.
- **Address Space Neutral:** Dynamic Privilege does not presuppose any specific address space design, allowing for various implementations in different address

[1] While unexplored in this dissertation, the author and collaborators conjecture that Dynamic Privilege offers a straightforward implementation in computer hardware by the addition of a single instruction to the ISA, and two bits to the per-task address space identifier.

[2] As seen in system call, interrupt return, and exception hardware.

space types, e.g., shared (like UNIX), separate (Microkernel), or even single address space OS designs (e.g. Opal, Mungi).

Integration with OSs

Dynamic Privilege can be integrated into existing operating systems or considered in the development of new ones. For instance, in a general-purpose OS like Linux, implementing Dynamic Privilege involves designing OS semantics for authorized threads post-elevation. This could include defining what an elevated thread can do beyond executing privileged hardware operations. Our prototype demonstrates this concept by adding support for Dynamic Privilege to Linux through a new system call. This enables elevated application threads to access and modify the OS kernel, effectively treating it as another dynamically linked library. However, the definition of Dynamic Privilege is not limited to this application. It could also apply to threads initially operating in the kernel, which can lower their privileges to restrict their operations, essentially creating a sandbox environment. This flexibility underscores that Dynamic Privilege does not tie itself to any specific OS architecture or application.

Figure 3·1 illustrates an example use of Dynamic Privilege, where a thread executes with hardware privilege for a sequence of instructions, then without, and finally with hardware privilege again.

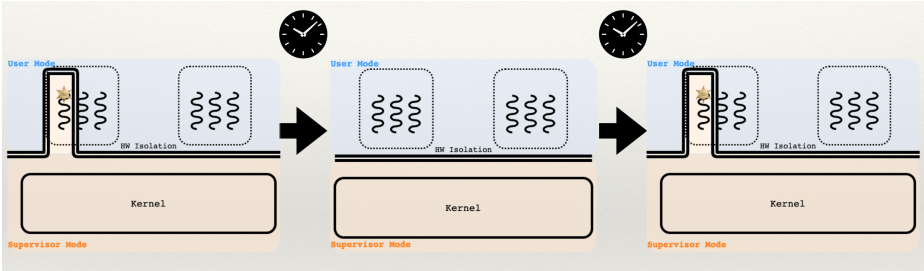

Figure 3·1: A thread of a process running on a general-purpose OS toggles between hardware privilege, from supervisor mode to user mode and back again.

Use Cases

Now we explore the definition by considering some brief illustrative examples of its realization and use. We intentionally defined Dynamic Privilege to separate mechanism from policy. By this, we mean the mechanism that realizes Dynamic Privilege within an OS need not have a bearing on the various policy decisions of the software that may manage it or come to use it. To that end, system and user software may utilize Dynamic Privilege to institute appropriate policies, treating hardware privilege as a resource. For example, a system may introduce a "Static Privilege" policy, where a thread is elevated to hardware privilege at creation time and remains elevated for its entire lifetime. Alternatively, a system may introduce a "Just-in-Time Privilege" policy, where a thread is elevated to hardware privilege only when it is about to execute a sequence of code that requires it, or perhaps a "Lazy Privilege" model where a thread executes at user privilege until it encounters a fault, at which point a software monitor determines the cause and whether the thread should be allowed to re-execute the faulting instruction with hardware privilege. These policies are orthogonal to the Dynamic Privilege mechanism, which simply enables the transfer of hardware privilege.

Given that the definition of Dynamic Privilege is OS-agnostic, a realization of the model will need to determine how it will relate to the systems security model. We discussed this in Chapter 1 Section 1.6 in the context of our Linux implementation, kElevate, in Section 3.2 of this chapter. In short, we argue that when realizing Dynamic Privilege on a traditional general-purpose OS, the use of the mechanism can be guarded by standard OS access control mechanisms, such as capabilities. This closely resembles the approach taken by the Linux kernel module system, which allows for arbitrary kernel modification, but restricts this ability to the credentialed administrator.

1.2 Access

Accessing and preventing access to computational resources is a long-studied area of computer science. General-purpose OSs are full of access control mechanisms spanning the spectrum from user-level controls,[3] to file system, process management, and system-level controls,[4] to hardware-level controls.[5] We define Dynamic Privilege to operate at the level of hardware execution modes. The choice of where to introduce Dynamic Privilege hinges on a tension between two primary concerns: the "scope of effect" and the "level of abstraction" for a given intervention. Instead of picking a middle ground to trade-off between these two concerns, we opted for the lowest possible level of abstraction and the largest possible scope of effect. Furthermore, as stated above, this ensures that Dynamic Privilege can be realized in an OS-agnostic way. This is made possible by the fact that 1) hardware privilege is "beneath" the level of the OS, and 2) hardware privilege modes are built into all general-purpose processors. This is in contrast to the other higher-level choices that would be OS-specific. The challenge then becomes how to build up from this low-level intervention point to provide the programmer with the desirable high-level application and OS abstractions and interfaces they need to do their work. Marrying the application and OS worlds is the focus of Chapter 4.

Consider that mechanisms like application permissions and file system permissions, for example, provide interfaces at a high level of abstraction: they are easy for programmers to understand and use. However, these high-level mechanisms offer limited scope for affecting the operating system's core components. For exam-

[3]These include User Groups and Roles, Users and the Superuser, Application Permissions, and User Interface Policies.

[4]File System Permissions, Process and Task Management, Containerization, C-Groups, Namespacing, Seccomp, Network Access Control, Kernel Modules, Device and Driver Access, and Credential Systems are some examples.

[5]Such as Virtualization and Isolation Controls include Virtualization Extension Access, Memory Access, Instruction Execution, and Hardware Privilege Levels.

ple, file system permissions allow a programmer to control access to files but not to modify the file system itself (much less reimplement the scheduler). Conversely, low-level mechanisms, like the ability to read, write, or execute kernel memory, acquired with kElevate allow extensive influence over larger operating system domains but are incomprehensible to most programmers.[6] In Chapter 4, we examine our layered approach to building useful semantics for elevated threads in a user process of a general-purpose OS, using our Linux prototype. We specifically show that Dynamic Privilege interacts well with the structure and facilities of Linux. This interaction enables the construction of compilation procedures, libraries, and tools that bridge the gap between the raw and powerful hardware privilege acquired with kElevate and the meaningful exploitation of the Linux kernel when elevated.

1.3 Revisiting the Source-to-Runtime Spectrum

Traditional OSs split user and kernel software between two distinct domains as discussed in the "source-to-runtime spectrum" (explored in Section 2.1.1). Separating these two domains precludes the type of optimization employed by Unikernels as discussed in Section 2.2.3. Developers of split-level OSs (like Monoliths, Microkernels, and Exokernels) design kernel code to utilize both privileged and unprivileged hardware capabilities, whereas user code remains limited to unprivileged capabilities. Kernel software provides a fixed entry point (the system call handler) that user code explicitly invokes, **simultaneously** switching hardware privilege and transferring flow control to the kernel domain.[7] [8] Developers use distinct build systems to

[6] Consider the Kprobes mechanism from Section 2.2.5, which can introduce code at the beginning or end of almost any kernel function. While this allows modification to just about any part of the kernel, it may not be practical to most systems programmers without high-level tooling like e-BPF.

[7] For example, the x86_64 instruction, `syscall` sets both the hardware privilege mode bits, and sets the program counter to the address (typically the system call handler) in the privileged `LSTAR` register.

[8] Kernels also perform similar simultaneous switching at typically implicit interrupt and exception points.

create a single privileged kernel binary (and modules, see Section 2.2.2) and a diverse array of unprivileged user libraries and executables. Runtime domains distinguish between privileged and unprivileged execution. "User runtimes,"[9] designated for user binaries, consistently disable the hardware privilege bit during execution. In contrast, the system sets up a "kernel runtime" during boot for kernel software, keeping the privilege bit always enabled during its operation. This static division of privilege and the coupled transitions between the two domains are fundamental to the traditional OS model; see Figure 3·2 for an example of these coupled transitions. Execution moves between application and kernel codebases, and simultaneously between user and kernel modes of execution.

Figure 3·2: Coupled transitions of traditional OSs. A transition between privilege levels (supervisor in orange and user in blue) is always accompanied by a transition between user and kernel runtimes. Atomic `syscall`, `sysret`, and `iret` instructions guarantee this.

[9] On Linux, the "user runtime" of interest is the thread executing within a process virtual address space. We use the term "user runtime" to generalize across the various user execution domains of different OSs, such as "events" within event-driven systems, "actors" in some distributed systems, and the many user-thread models such as co-routines and green threads.

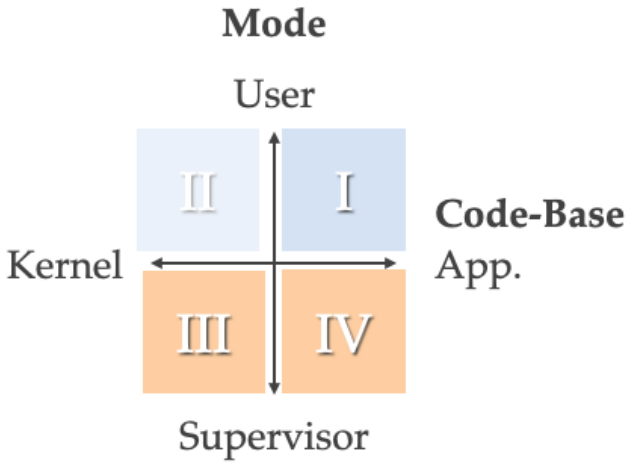

Figure 3·3: Traditional bi-level OSs operate within quadrants I and III of this execution mode x codebase graph, carefully coupling transition between privilege levels and runtimes. OSs implementing Dynamic Privilege can operate in all four quadrants, though this dissertation focuses on toggling between quadrants I and IV: application codebases running with and without supervisor privilege.

Dynamic Privilege introduces a new degree of freedom by decoupling hardware privilege transfer from control flow transfer. It enables runtime execution to switch privilege on demand while maintaining execution within the same domain. A subsequent invocation of kElevate revokes the privilege. This decoupling paves the way for innovative runtimes, such as elevated user runtimes with hardware privileges. As illustrated in Figure 3·3, Dynamic Privilege facilitates the exploration across all four quadrants of the execution mode x codebase graph. In Chapter 4, we will show that introducing Dynamic Privilege into Linux lets us build hybrid applications that eliminate the boundaries between application and OS in controlled ways. We quantify active optimizations in Section 4.2.

2 The kElevate Mechanism: Implementing Dynamic Privilege on a General-Purpose OS

This section outlines our approach to introducing Dynamic Privilege into an existing general-purpose operating system. Our objective is to facilitate application access to hardware privilege through a fine-grained execution mode transfer mechanism. This mechanism, named kElevate, enables user code to operate with enhanced hardware privileges, something outside the design of our target OS, Linux.[10]

The kElevate implementation does not aim to provide comprehensive semantics for what actions an elevated thread can perform. Specifically, it doesn't guarantee that an elevated thread will continue functioning as a normal user thread would without causing crashes to itself or the system. Our strategy for implementing Dynamic Privilege semantics is incremental and layered. For instance, if the requirement is limited to executing privileged hardware instructions for reading system registers, then kElevate's functionality is adequate. However, if one intends to call an arbitrary kernel routine post-kElevate, this is typically not "safe," as it may lead to crashes of the process or system. We focus on building layers of libraries and tools that, following an initial kElevate call, adapt the process or system to enable various elevated operations with well-defined functionality. This approach allows the Dynamic Privilege semantics for Linux to evolve and develop as necessary, instead of being confined to an arbitrary predetermined set.

In Chapter 5, we chronicle the set of "adaptors" we developed, which are sufficient for support executing many rich programs. We know our set of adaptors is insufficient for executing all programs elevated, (Firefox is one that crashes, for instance). We

[10]While it is not hard to imagine how one could exploit existing Linux kernel paths to permet kernel threads to lower their privilege, this is beyond the scope of this dissertation. We address this in the future work section of Chapter 6.

encourage future work to explore the development of additional adaptors to support more programs.

While one could design a new hardware privilege switching instruction, we have chosen to focus on the implications of having the ability to switch privilege and simply prototype the mechanism as a system call in Linux, effectively emulating such an instruction. In the following subsection, we will discuss how our modification uniquely enables the conditional continuation of supervisor mode execution past the conclusion of a system call, a property intentionally avoided in conventional operating system designs.

2.1 Implementing kElevate

We have prototyped kElevate as a new system call in Linux for x86_64 and ARM64 platforms. This system call is designed with conditional exit logic that (when requested) does not revoke hardware privilege after execution, allowing a thread to persist with elevated privileges. Following the system call, the subsequent instructions execute in this "elevated" mode of hardware privilege. The system call accepts an argument specifying the desired privilege level upon return and whether interrupts are enabled.

Our implementation demonstrates the simplicity of integrating kElevate into the Linux kernel. This experience leads us to believe that similar integration will be feasible for other general-purpose OSs. Because kElevate operates so close to the hardware level, it did not interfere with much of the higher-level OS functionality. The following chapter, Chapter 5, addresses some paths that were affected by our change, and how we addressed them using "adaptors." We implemented kElevate on both x86 and ARM because these two architectures represent a large portion of the computing landscape. The x86_64 implementation required only 173 lines of code changed (after removing comments and white space changes). The straightforward

nature of our implementation suggests that kElevate could be implemented on other architectures as well.

Figure 3·4 illustrates the standard Linux virtual address space layout highlighting privilege. In contrast, Figure 3·5 illustrates the virtual address space layout after a call to kElevate has been completed. Given that kElevate operates on a per-thread basis, the view illustrated is specific to the calling thread.

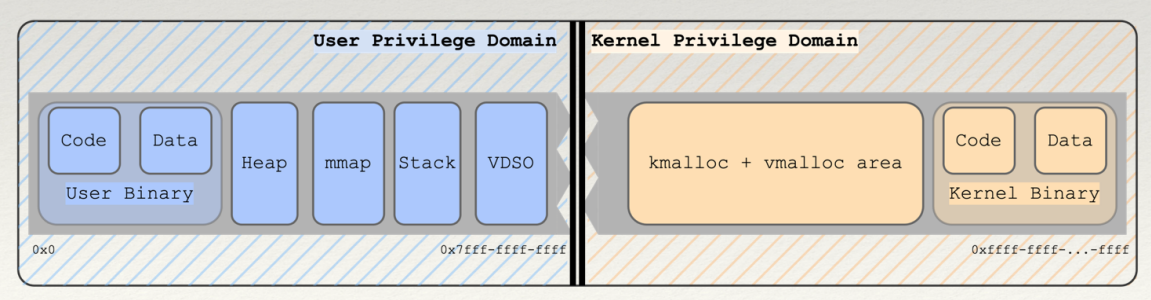

Figure 3·4: A lowered Linux Process Virtual Address Space. The MMU prevents user thread access to the upper half of the virtual address space where the kernel resides.

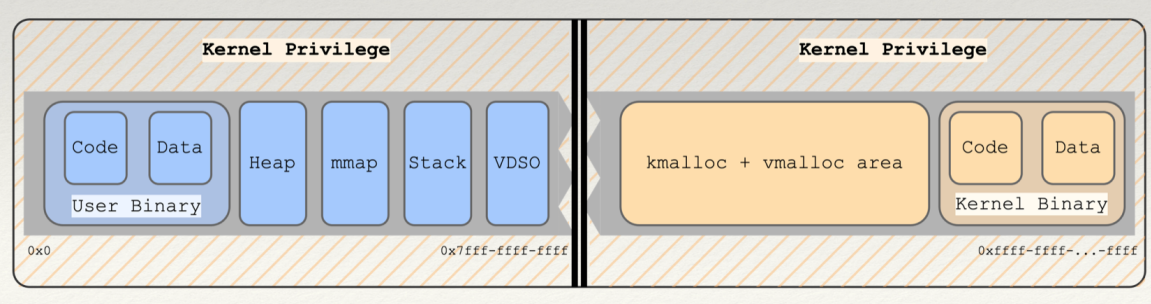

Figure 3·5: An elevated Linux Process Virtual Address Space. User threads are granted access to the upper half of the virtual address space.

2.2 kElevate implementation

Implementing the kElevate mechanism in the Linux kernel required some changes to the Linux kernel at version 5.14. We will describe the x86_64 implementation in detail and then discuss the few differences in the ARM64 implementation.

The complete set of Linux kernel changes are as follows:

- Added the `kElevate()` system call.
- Adding a `KELEVATE` kernel configuration option, providing control over whether the `kElevate()` system call is included in a given kernel build. Corresponding `CONFIG_KELEVATE` preprocessor macros were added to bracket every part of the `kElevate()` implementation.
- Added an 8-byte field to the per-thread `task_struct`. Our implementation currently uses two bits of this: one tracking current elevation status, and one tracking whether an elevated thread is undergoing OS migration to another core.
- On our x86 implementation, when migrating an elevated thread to a new core, we update the user version of the GS segment base register to the new core's per-core data structure.
- Adding a `CAP_ELEV` capability and checking that a thread has it before executing the body of the system call.
- Modified the system call return path, adding a conditional check of the elevated bit. When set, the thread simply stack switches and returns to the application in supervisor mode using the `ret` instruction.
- On our x86 implementation we updated the GS register to reflect changes in the new CPU state.
- At boot time, we disable the SMEP (Supervisor Mode Execution Prevention), SMAP (Supervisor Mode Access Prevention), and KASLR (Kernel Address Space Layout Randomization) security features. `kElevate`.

Thread Control Block Changes: In our x86_64 implementation, we introduced an 8-byte kElevateStatus field in Linux's version of the Thread Control Block (TCB),[11] a structure known as the `task_struct`. This field comprises two used bits: the isElevated bit[12] monitors the thread's elevation status, and the isMigrating bit monitors whether an elevated thread is undergoing migration to another core.

System Call Handler and Exit Paths Changes: Linux applications typically issue system calls with the `syscall` instruction and Linux typically returns from system calls using the fast `sysret` instruction, which are utilized to switch into and out of HW privilege execution mode respectively. We modified the common system call handler exit routine to check the isElevated bit, and if set, to return to the user with a stack switch and a simple `ret` instruction, which does not induce a change in privilege.

Thread Migration and GS: On x86 hardware, the Linux kernel utilizes the per-core base register corresponding to the GS segment register as per-core memory. This register is used to access per-core data structures, such as the current actively running TCB. Because core migrations are transparent to application threads, an application thread may be migrated to a new core at any time. This is a challenge if the application thread is using the GS register, as it will be stale after a core migration. While this change can likely be factored out to and adaptor (see Chapter 5, for now we have implemented this logic in the thread migration code. The isMigrating bit is set if a thread is migrating and isElevated is true. And when the migration is complete, the application's GS register base is updated if the isMigrating bit is set.

[11] This is not an exact mapping as the `task_struct` also contains the information of a Process Control Block.

[12] The reserved bits have the potential to accommodate multiple privilege levels in the future.

SMEP, SMAP, and KASLR: The three security features we disable at boot time are SMEP (Supervisor Mode Execution Prevention), SMAP (Supervisor Mode Access Prevention), and KASLR (Kernel Address Space Layout Randomization). KASLR is a software feature that randomizes the location of the kernel in memory, making it more difficult for attackers to exploit kernel vulnerabilities. SMEP and SMAP are hardware features that cause exceptions to occur if hardware privilege is enabled and attempts are made to execute or access memory marked as user.

Currently, we sidestep these issues by supplying the that the `nosmap`, `nosmep`, and `nokaslr` arguments are specified when the kernel is booted. Disabling these is undesirable because it erodes system security. Because KASLR only applies at boot time, it will be easy to circumvent because our dynamic linking approach (see Chapter 4) can recover symbol locations at runtime utilizing /proc/kallsyms.

To eliminate these the need for `nosmap` by disabling `SMAP` at runtime using elevated code, and modifying the corresponding bit in control register 4. Disabling `SMEP` is tricker because the kernel maintains a shadow copy of the boot setting, which we didn't pursue further. While context-switching control register 4 is a course-grained solution, ultimately we want to control the page table permissions of the elevated process to mark relevant pages as supervisor accessible/executable in a fine-grain manner.

Securing kElevate

As mentioned in Section 3.1.1, left unguarded, Dynamic Privilege may lead to security vulnerabilities. Linux's capability subsystem prevents unauthorized threads from executing certain system calls. The capability subsystem allows for the specification of which entities [13] are allowed to perform which actions or access which resources,

[13]Capabilities can be associated with many abstractions such as processes, threads, files, users, groups, and namespaces.

at a fine-granularity. Capabilities protect system calls like `systemctl()`, `mount()`, and `insmod()`. The only thing preventing an unauthorized user from executing these system calls is a conditional check on whether the corresponding entity (often the user) carries the corresponding capability (often `CAP_SYS_ADMIN`). If they don't, the body of the system call does not execute, instead returning `EPERM`, the "no permission" error.

kElevate follows this design pattern. The kElevate system call actively performs a conditional check when executed, ensuring that the calling process started from an executable endowed with the `CAP_ELEV` capability, which we added to the Linux kernel. This capability is not granted to any executable by default; it requires explicit assignment. Administrators can add this capability using the `setcap` command, a part of the `libcap` package, which is standard in most Linux distributions. Executing `setcap` necessitates the `CAP_SETFCAP` capability, available to the `root` user by default.

ARM64

We have focused primarily on developing and testing the x86_64 version of our implementation, but we have also created and tested a version for ARM64. This ARM64 implementation has hardware-specific differences and is somewhat simpler than its x86_64 counterpart. On ARM platforms, we don't need to add extra state in the Thread Control Block (TCB), as hardware privilege levels are managed through the processor register (`CPSR`), which privileged instructions can directly access.

In our ARM implementation, the `kElevate` system call handler modifies the permissions of the virtual memory page that contains the calling process's return address. Specifically, it configures these permissions to allow execution with kernel hardware privileges (`EL1`). To proceed with further execution, one must apply adaptors using elevated code, implementing additional modifications to the process's memory permissions (see Chapter 5).We are currently investigating Linux's existing support for setting these permissions before invoking `kElevate`.

3 Framework

We conclude this chapter with a section offering a three-part conceptual framework (illustrated in Table 3.1) to communicate the impact of implementing the Dynamic Privilege OS model on the application domain of a general-purpose OS. A core contribution of this part of our work is introducing OS-level access to the application domain without compromising its other qualities. We use this framework to discuss the implications of our work in the context of OS extension models and our past work on Unikernels, including EbbRT and UKL.

Dynamic Privilege uniquely upholds three core properties when blending application and kernel codebases: procedural control, OS-level access, and full compatibility with the application execution model.

	Linux Process	Extension	UKL	Process w/ kElevate
App. Control	✓	X	✓	✓
Kernel-Access	X	✓	✓	✓
App. Exec. Model	✓	X	∼	✓

Table 3.1: Comparison between Linux Processes, classical OS extension techniques, a modern Unikernel: UKL, and a Linux process with access to hardware privilege via kElevate. Linux applications have procedural control of their execution and access to the rich software ecosystem conferred by the application execution model but no OS-level access. Classical OS extensions rely on registering handlers into the OS framework, inverting control, and accepting the restricted kernel model. UKL buys back control but compromises between the application and kernel execution models. kElevate claims all three.

3.1 App. Control and Framework Extension

OSs have traditionally allowed for runtime kernel modification, or extensibility, using the framework extension approach. In this approach, users register handlers (typically implemented at architected hook points) to be executed when the framework invokes them. When applied to kernel extension, this approach enables the creation of specialized paths and the re-implementation of kernel internals.

Framework extension is intentionally limited by design. It effectively hands over procedural control to the framework (in this case, the kernel). The framework serves as an external event loop, invoking the handler when encountering a relevant hook. This leads to an inversion of control (IoC) (Gamma et al., 1994), discussed later in this subsection.

Put simply, a framework is a software template with intentional hook points for adding custom extensions. Systems researchers may recognize this design in systems like Dracut, Systemd, and Emacs, which feature hooks for adding custom modules, service definitions, and activation hooks. FileSystem in Userspace (Contributors, 2020) (FUSE) similarly offers this framework model, providing hooks for file operation callbacks like open, read, write, etc.

Extending this analogy to OS kernels, maintaining kernel integrity was paramount when computers were costly, centralized, and tasked with handling multiple users' code. Past work on OS extensions—whether by incorporating application code or reimplementing kernel components—focused on preserving the integrity of the kernel framework and its resources like data, CPU, and locks. Regrettably, even with all these trends shifting, more recent systems have not significantly advanced the power of extension mechanisms.

One popular method for extending Linux involves adding kernel modules (Kernel-Contributers, 2022), which interface at designated kernel points to implement subsystems like filesystems, virtual filesystems, and device drivers. These modules register callbacks, functioning in an event-driven model and exhibiting the same IoC as suffered in application frameworks. e-BPF (Contributors, 2016b) handlers are a nuanced example of this, providing some static analysis and often installed at non-architected hook points.

Object-oriented OSs like EbbRT(Schatzberg et al., 2016), Tornado(Gamsa et al.,

1999), Apertos (Itoh et al., 1995), u-Choices (Campbell and Tan, 1995), Vino (Small and Seltzer, 1994), and k42 (Krieger et al., 2006) also exhibit IoC. While their object-oriented design facilitates the replacement of objects using inheritance and polymorphism, the kernel framework is still the engine determining when and which objects will be executed.

In contrast to the above extension approaches, UKL links an entire process into the Linux kernel and breaks from the tradition of IoC, allowing optimized applications to retain direct procedural control over their execution. kElevate provides an application thread with the ability to execute with hardware privilege. Standard Linux application threads have control of their execution; nothing about that changes when they take on privilege using kElevate.

IoC is not an inherently detrimental design pattern; multiple of our tools make use of it. However, an OS solely permitting IoC-based extensions lacks flexibility compared to one that accommodates both IoC and standard procedural control because it increases friction for developers looking to deviate from the kernel framework's structure. By contrast, Dynamic Privilege allows for scripting, which we exploit with Makefiles, Bash scripts, and Python scripts. Instead of the kernel driving extension execution, the programmer can control the procedural execution of potentially many privileged executables developed with the UNIX philosophy: "Make each program do one thing well."

3.2 Kernel-Access and SDKs

Traditionally, applications were restricted to a black-box system call API in Monolithic OSs, unable to modify core OS abstractions like files, processes, or address spaces. This limits hardware access (like writing directly to a device) and hampers cross-boundary compiler optimizations (like those discussed in Section 4.1.5). While enhancing security and system structure (applications arbitrarily replacing privileged

system components can quickly lead to exploitation), this model stifles holistic optimization (as done in Unikernels) and limits developer freedom (like the ability to introduce specialized kernel paths).

In contrast, however, powerful extension mechanisms like Linux's kernel modules have significant ability to access the kernel's internals. Kernel modules can use the same types and macros resolved through the build system, and exported kernel functions serve as an interface that can be dynamically linked to at module insertion time. The kernel module subsystem is essentially a Software Development Kit (SDK), providing low-level access for the developer to recombine and reimplement system code paths. UKL gains OS-level access instead by statically linking an application into the kernel—the linker resolves symbols ahead of time (static and dynamic linking are discussed at length in Section 4.1.6).

While kElevate grants only hardware access, we demonstrate that it is a sufficient foundation for achieving OS-level access. We utilize two tools to accomplish this: the kernel module build system and a shared library generated from the kernel's System Map–equivalently, /proc/kallysms. First, a slightly modified version of the kernel module subsystem compiles new source files into object files utilizing OS-level types, inline functions, and macros. These alterations to the module build system produce objects capable of integration into an elevated process's compilation. Second, when running with hardware privilege, the kernel side of the process's address space becomes accessible. This is similar to how a standard process gains access to its address space after a shared library is dynamically loaded. An elevated process can thus link with a shared object, resolving symbols at runtime through a Procedure Linkage Table (PLT), a standard tool used in dynamic linking. This is enough to provide SDK-level access to the kernel.

3.3 Execution-Models and Client-Server Decoupling

The well-known Client-Server model offers a critical decoupling between the implementation of the two components. The two sides merely agree to a shared protocol, and the rest is left to independent development choices. Perhaps the client is written in Python and utilizes many long-lived threads, while the server is implemented in C++ and uses a run-to-completion and event-driven model. Both components liberally include libraries to reduce programmer effort.

Analogously, consider a Linux process as a client and the kernel as a server. The two agree on the system call interface. They are free to enjoy a similar decoupling, with the process using any language, any build system, and any execution model. At the same time, the kernel is implemented in C using Make and running in a heavily restricted environment. The kernel disables certain hardware optimizations and builds with a large set of special compiler flags, many of which conflict with those used to build applications.[14] Most importantly, reducing the kernel attack surface requires the kernel's execution model to supply a limited set of libraries. This may be the single greatest departure from the application model of execution.[15]

Systems integrating applications or components into the kernel must adhere to the kernel's execution model, effectively locking down parameters like language, build system, compiler flags, and library access. Returning to the Client-Server analogy, this constitutes a coupling where part or all of the client must be brought into agreement with the server as preparation for insertion into the server.

At the risk of overemphasizing this point, the difficulty here is not merely in actualizing the all-to-one mapping of every type of application[16] to an arbitrary model.

[14]Some examples are: the kernel uses a custom linker script, kernel code cannot be built with the flags for standard position independence, the use of Red Zones, and must be built for execution in the upper 2 gigabytes of the virtual address space.

[15]An informal count shows 10 GitHub repositories for every *line of code* in the Linux kernel.

[16]Recursively including all of its libraries.

It is the choice of *image*–the output of the mapping–to be the kernel itself. The kernel is a pathological choice in how alien it is to the application execution model and how intentionally restricted it is.

Processes that run with privilege execute in the same part of the virtual address space that Linux intended to execute any process, and they can preserve ABI compatibility. They need not move code into the kernel. We have multiple examples demonstrating the degree of application compatibility we uphold. For instance, elevated processes are compatible with containerization with unmodified application binaries, and we have implemented examples extensively in Python and C and, to a lesser extent, in Assembly, C++, Rust, Go, and Java. Further, using the kernel module system to act as a preprocessor and our dynamic linking approach to resolving all kernel symbols, we have been able to utilize or modify core system components like memory allocators, page table code, interrupt handlers, as well as the Interrupt Descriptor Table (IDT) and extensively exploit the system call handing paths.

Chapter 4

Case Studies

Dynamic Privilege enables programmers to do things they couldn't before. This dissertation largely explores what becomes possible when applications can access new parts of the systems and Instruction Set Architecture (ISA).

1 Hands-on kElevate

This section organizes a list of capabilities that Dynamic Privilege brings to elevated processes and offers examples demonstrating each capability. We categorize these capabilities into 1) low-level hardware access, 2) OS-level access, and 3) other elevated process properties. These examples flesh out the three-part framework (OS-access, flow control, and the application context) provided in Section 3.3. This section focuses on notions of access because this is the new property Dynamic Privilege brings to elevated processes. These examples demonstrate how we bridge the gap between application and kernel worlds by building tools exploiting the full-featured application context and the kernel's power.

Low-level hardware access: The following exemplars demonstrate that the kElevate mechanism can toggle a user application thread into the privileged or supervisor execution mode. From there, developers can exploit privilege for bracketed periods, mixing privileged instructions into their application binaries. Elevated processes can read and write system registers and all kernel memory, as well as execute all supervisor

instructions.

OS-level access: In addition to the raw ability to execute with hardware privilege, we build tools to allow elevated processes to exploit all the types, macros, data structures, and subroutines of the kernel. Access to types, macros, and inline functions is done using the provided Linux kernel module subsystem, which utilizes the kernel headers. Access to kernel symbols, including data structures and standalone functions, is achieved through dynamic linking to a shared library we produce using the kernel's `kallsyms`(SRL, 2005) utility. We have worked to expose this tooling through standard shared objects, with the goal of compatibility with any language runtime that supports dynamic linking shared libraries. We have demonstrated compatibility with multiple languages, including C, C++, Python, and Rust.

Exposing our incrementally growing body of Linux-elevated process code as standard dynamic libraries has let us mix the power of system programming with user-centric high-level languages and their ecosystem of packages. To highlight this, the examples in this section utilize the integration of elevated code in the interactive Jupyter/IPython Notebooks(Pérez and Granger, 2007) to illustrate the basic capabilities of elevated code use. These examples emphasize functionality and how this eases programmer implementation efforts.

This section demonstrates the following capabilities of elevated processes:

Low-Level Hardware Access

1. Bracketing privilege access in time.
2. Executing privileged instructions from user code.
3. Accessing privileged registers from user code.
4. Accessing hardware data structures directly.

OS-Level Access

5. Executing kernel subroutines from user code.
   - 5.a. Standalone kernel functions
   - 5.b. Inline kernel functions
6. Accessing kernel data structures from user code.
7. Utilizing kernel types.
8. Utilizing kernel macros.
9. Dynamic linking elevated process to kernel symbols.

Other Elevated Process Properties

10. Full-Featured Application Context
11. Retaining flow control.

Figure 4·1: Dynamic Privilege provides these capabilities within elevated processes. As such, privileged processes can utilize code designed for any level of abstraction from the hardware-level, to the OS-level, to the application context.

This section proceeds with the following subsections: Section 4.1.2 demonstrates reading a privileged register, Section 4.1.3 demonstrates accessing hardware data structures, Section 4.1.4 demonstrates executing kernel code paths, Section 4.1.5 demonstrates using kernel types, macros, and inline functions, Section 4.1.6 demonstrates dynamic linking elevated processes to the Linux kernel.

1.1 Principles of Dynamic Privilege Integration

Before jumping into our first example, we take a moment to highlight some basic properties of Dynamic Privilege, especially where it makes contact with the standard software toolchains we use throughout these examples. The following examples illustrate that when Dynamic Privilege is added to an existing OS, it naturally integrates and is compatible with the existing user and kernel software and tooling. Our examples use standard source code, user libraries, and toolchains. We build toward exposing the running kernel as just another dynamic library. This way, an application can link to the kernel without sacrificing its personality as a standard user process.

To our knowledge, no existing general-purpose OS mechanism permits a user instruction stream to execute a privileged instruction directly without triggering a fault or altering flow control to kernel code (implicitly via a trap or explicitly via a system call). Dynamic Privilege allows privileged instructions to operate with zero domain transition overhead, preserving CPU execution optimizations such as caching, pipelining, branch and value prediction, speculation, etc.

Since standard toolchains, including compilers and linkers, do not differentiate between privileged and non-privileged instructions,[1] designing, implementing, and integrating code sequences with both instruction types into object modules for linking into application processes is straightforward. No additional modifications or tooling are necessary to harness Dynamic Privilege. The underlying reliance of Dynamic Privilege on standard instruction specifications and execution ensures its compatibility with all conventional application runtimes and source methodologies, such as compiling, shared objects, position-independent code, and dynamic link loading.

[1] GCC builds both the Linux kernel and user-level applications. GCC will happily build a user executable that includes privileged assembly instructions without so much as a warning. Without Dynamic Privilege, the OS will catch the issue at runtime, segmentation faulting the process.

1.2 Reading a Privileged Register

Privileged registers are protected from user access to prevent system modification and compromise. This example shows that elevated processes can access them and demonstrates other forms of low-level hardware access from the application context. Control Register 3 (CR3) on x86_64 processors is such a register holding the physical address of the currently loaded page table root. When an unprivileged thread attempts to execute a `mov cr3, %rax` instruction, the processor raises a general protection exception, and the kernel kills the process. However, direct CR3 (and thus page table) access from the application context could benefit a system programmer. It could facilitate the construction of memory management tools unattainable in user mode, such as high-performance garbage collectors (Belay et al., 2012) and process checkpoint and replay (Cadden et al., 2020).

Dynamic Privilege Properties Exemplified

This first example, Figure 4·2, captures multiple facets of our use of Dynamic Privilege via the kElevate mechanism, including items 1-3 and 10 in Section 4.1 above. This example shows that Dynamic Privilege's runtime property allows for "bracketing" the duration and number of instructions permitted to execute with privilege. This indicates that Dynamic Privilege is not a binary system-wide property; not all processes or binaries need to be granted privilege, and developers can restrict the sequence of instructions and the timeframe of elevated privilege execution. Bracketing offers a degree of defensive programming, limiting the exposure for errors like wild writes to kernel memory that could take down the system. Changing privilege levels does come at the cost of a system call.

```
1 priv = loadMod(priv_lib_path)
2 kele = loadMod(kele_lib_path)
3 kele.kElevate()
4 cr3 = priv.getcr3()
5 kele.kLower()
6 print(f"CR3 = 0x{cr3:x}")
```

```
1          CR3 = 0x2e868002
```

Figure 4·2: This example demonstrates: 1) bracketing an elevated code block with entering and exiting privilege; 2) the execution of a privileged instruction; and 3) accessing a privileged register. Don't overlook that this is all done from 4) the application context in a high-level language, Python, and in an interactive, exploratory environment, Jupyter.

Step by Step

Lines 1 and 2 of Figure 4·2 involve loading Python modules we developed using the standard infrastructure for creating dynamic shared objects (DSOs) from C code. The Python runtime can execute functions in these modules via generated trampoline routines that establish the necessary Python language entry points. At the core of these DSOs are C functions designed to 1) Invoke the new kElevate system call, and 2) define a function that includes the privileged instruction that moves the value in the x86 CR3 register into a general-purpose register, returning that value.

Line 3 elevates the privilege level of the Python runtime, ensuring that all subsequent operations are performed with supervisor privilege. Thus, the function executed on line 4, which runs a privileged instruction, completes without causing a processor trap. The kLower call on line 5 reverts the Python process's privilege level back to user mode. Finally, line 6, operating without supervisor privilege, outputs the privileged information obtained on line 4. The `getcr3()` call validates that the elevated process can perform the `mov cr3, %rax` instruction, and the subsequent output confirms that kElevate facilitated access to privileged hardware state. Rerunning the example without the elevation call leads to a segmentation fault.

Discussion

Elevated processes encourage system and kernel prototyping, exploration, and communication through approachable, interactive, and user-friendly formats. Demonstrated in Python within a Jupyter Notebook, it contrasts with traditional kernel development cycles involving module building at best or kernel recompilation at worst. The kElevate mechanism, coded in C and integrated into the Python interpreter via standard Linux shared objects, exemplifies Dynamic Privilege's compatibility with rich application contexts. Once familiar with privilege elevation, Python library authors can utilize their understanding of the hardware to create complex operations mixing privileged and non-privileged actions. We believe that opening the kernel to this type of high-level manipulation will benefit research, industrial, and educational settings. We have rewritten and successfully executed this example in Assembly, C, C++, Rust, Go, Java, and Python. While this example works without caveat, more sophisticated tools may require kernel "adaptors," discussed in section 5.

1.3 Accessing Hardware Data Structures

Modern hardware defines various data structures that software (particularly OSs) can use to control the system's runtime behavior. This example demonstrates using hardware privilege to modify the x86 Interrupt Descriptor Table (IDT) data structure. While later examples will show that elevated processes can meaningfully interact with the underlying OS, operating directly on hardware structures is interesting because it enables building tools, such as debuggers and performance monitors, that operate independently from the OS implementation. These may be particularly useful for creating OS-agnostic tools, such as debuggers and performance monitors.[2] While mature OSs may still benefit from their custom uses of monitoring hardware (like

[2]Modern processors have complex performance monitoring hardware, including subsystems like Intel's Performance Monitoring Counters, VTune, Cache Monitoring Technology, and Process Trace.

Linux's use of Perf), OS-agnostic tools offer an approach to "factoring out" a core set of functionality that applies across OSs. Custom OS developers would likely appreciate not having to develop their own monitoring tools (as we did with EbbRT). The IDT is one of the most fundamental data structures on x86 hardware; it controls where control flow goes when an interrupt occurs and configures how the interrupt is handled (e.g., what stack it runs on). While details of what this modification achieves are explained in Chapter 5, this example accesses the IDT through repeated execution of the child elevated process as a Python subprocess, reading its contents into high-level data science tools, a Pandas DataFrame.

```
1  def read_whole_idt ():
2    import pandas as pd
3    df = pd.DataFrame (columns=["full addr:", "segment:", "ist:"
       , "zero:", "type:", "dpl:", "p:"])
4    for i in range (IDT_ENTRIES) :
5      output = subprocess.check_output ([symbios_path + "Tools/
         bin/idt_tool", "-p", "-v", str(i)])
6      get_all_desc_values(df, output)
7
8      return df
```

```
9  old_idt = read_whole_idt ()
10 run_cmd (symbios_path + "Tools/bin/recipes/mitigate_all.sh")
11 new_idt = read_whole_idt ()
```

```
12       mitigated core 0
13       mitigation finished
```

Figure 4·3: Example: Using high-level language tools to parse and analyze hardware-defined data-structure. This example uses an external command to run the Elevated Code. We are currently extending the Elevated Code library to make this functionality directly available in Python. Specifically, it creates a copy of the IDT and then modifies it.

```
14 # We see that len (modified entries) = 3, and the indices
       correspond to the
15 # double fault, text fault, and debug trap adaptors.
16
17 # Here, we check which entries of the IDT have changed.
18 modified_entries = [i for i in range(IDT_ENTRIES) if not
       old_idt.iloc[i].equals (new_idt.iloc [i])]
19
20 print "Modified entries: ", len (modified_entries),
       modified_entries)
```

```
21       Modified entries: 3 [1, 8, 14]
```

```
22 for i in modified_entries:
23   print("Entry ", i)
24   print(old_idt.iloc[i])
25   print (new_idt.iloc [i])
```

```
26 Entry 1
27 full addr: 0xffffc90001a4f000
28 ...
29 Entry 1
30 full addr: 0xffffc90001a9f000
31 ...
```

Figure 4·4: Example: Using high-level language tools to parse and analyze hardware-defined data structure. This example uses Python code to detect changes in the IDT modifications enacted in the prior example.

Step by Step

Reading and modifying the IDT: The first figure, Figure 4·3, illustrates defining (line 1), then using (lines 9 and 11) a Python function read_whole_idt to read each of the vectors of the x86_64 IDT before and after an intervention that modifies it. This function employs the Pandas library to create a DataFrame for storing IDT entries. It iterates over all 256 IDT entries, executing an external privileged process (idt_tool) for each entry on line 5. The output of this command is processed by get_all_desc_values (line 6) to populate the DataFrame. The second listing uses this function to read the IDT into a variable old_idt on line 9, executes a script to modify the IDT (line

10), and then reads the modified IDT into new_idt (line 11). This captures the full state of the IDT hardware data structure before and after a modifying procedure. We will elaborate on the mitigate_all script in Chapter 5, but mechanistically, it deploys about a dozen elevated processes per-core in parallel, modifying the IDT. On our system with 84 cores, this amounts to about a thousand elevated processes in total.

Analyzing the IDT Modifications In the analysis phase (Figure 4·4), the Python code locates the specific modifications made to the IDT. Initially, in line 18, the script determines the modified IDT entries by comparing each entry in the original (old_idt) and the modified (new_idt) IDTs. It then prints, as shown in line 21, the total number of modified entries along with their respective indices. Comparing the two DataFrames, we see the full scope of changes to the IDT; vectors 0, 8, and 14 have changed, corresponding to the debug, double fault, and page fault handlers, respectively. Following this, a loop starting from line 22 iteratively prints the details for each modified entry. Taking a closer look at lines 27 and 30, we can see that the address of the debug handler has changed.

Dynamic Privilege Properties Exemplified

This example reads, modifies, and rereads the IDT, demonstrating capability 4, accessing hardware data structures. Further, it kicks off hundreds of elevated processes, one per line of the IDT, and a thousand more running in parallel across many cores during the execution of `mitigate_all.sh`. This demonstrates capability 11, retaining control flow; the script determines when the elevated processes run, and this choice is not delegated to the kernel framework. Further, the parallel execution of these elevated processes demonstrates something that Unikernels cannot do: running multiple elevated processes. Finally, utilizing high-level data science tools like Pandas DataFrames demonstrates property 10, that we can use high-level tooling available

in this full-featured application context. Students and kernel developers can discover and evolve systems using these familiar tools and workflows.

Recall the framework of Chapter 3 Section 3.3. This example shows properties of control flow and the full-featured application context. This example reads the IDT directly as a subroutine, preserving flow control. Contrast that with the indirection experienced reading this information from an e-BPF map filled when a handler runs, or through an engineered `/proc` filesystem interface. Further, consider that this example is packaged as a standalone program that can be invoked in parallel by driving scripts, all part of building out a full-featured application context.

1.4 Executing Kernel Code Paths

A primary goal of this dissertation is to investigate the feasibility and methods of utilizing Dynamic Privilege to enhance the utility of kernel code paths through integration with user-level software. Perhaps surprisingly, we have found that we can use the Linux kernel subroutines similarly to how one would use a standard application library. This approach echoes the way applications interact with LibraryOSs in Unikernel architectures. We speculate this is due, in part, to the inherently modular design of the Linux kernel, a common strategy employed in complex codebases to manage and reduce complexity.

General-purpose OS kernels like Linux are some of the largest software projects, with close to 30 million lines of code, and 1.2 million commits(Github, 2023). The Linux kernel is engineered to separate common code paths from architecture-specific implementation. Further, it is highly configurable, supporting over 150,000 options(Mathieu Acher, 2019) as of 2019. The kernel is organized into subsystems and procedurally organized with reuse in mind. This modularity is a key enabler of our approach. We can use the kernel as a library in our elevated processes because the kernel is designed to be used as a library by kernel developers.

Dynamic Privilege Properties Exemplified

One step toward using the kernel as a library is to execute kernel code paths from user code. The two examples in this section explore item 5a from Figure 4.1, executing standalone kernel functions. Figure 4·5 shows an example of executing a standalone kernel function from an elevated process, the Python interpreter. `getSymAddr()` is a wrapper for the kernel function `kallsyms_lookup_name`. Given its name, this function returns the address of a kernel symbol, such as a standalone function or variable. It is a useful kernel function when one needs to find the address of a kernel function; it takes a symbol as a string and returns the corresponding text address in the kernel. In Figure 4·5, we use it to find `ksys_write`'s address and verify it against the kernel's system map file created during kernel build.

```
kele = loadMod(kele_Lib_path)
ksys_write_addr = kele.getSymAddr("ksys_write")
print("ksys_write: 0x%x" % ksys_write_addr)
```

```
    ksys_write: 0xffffffff81367220
```

```
import os
os.system("cat /boot/System.map-5.14.0-kelevate
   | grep ksys_write")
```

```
    ffffffff81367220 T ksys_write
```

Figure 4·5: Example: Executing an internal kernel routine within a python process. The `getSymAddr` python function exploits Elevated Code that invokes the internal kernel routine `kallsyms_lookup_name` to determine the address of the kernel function `ksys_write`.

Our next example is again a demonstration of property 5a. `ksys_write` contains the implementation of the `write` system call handler. It has the same signature as the C library function `write`. We demonstrate intermixing writes to a file descriptor using the C library and kernel functions. By printing the file's contents, we demonstrate the intermixing of these code paths.

```
with open("output.txt","w") as f:
f.write("Hello")
kele.kElevate()
kele.ksys_write(f.fileno(), "World!\n, 7")
kele.kLower()
```

```
with open ("output.txt", "r") as f:
print(f.read ())
```

```
    Hello World!
```

Figure 4·6: Example: Calling an internal kernel system call handler within a python process.

Discussion

Calling internal kernel code paths directly from the application context is an unusual blending of application and kernel programming worlds not explored outside the Unikernel literature. Directly invoking system call handlers, so-called "shortcutting," is one technique we introduced and explored in UKL. We implement shortcutting in a different way using Dynamic Privilege, discussed in Section 4.2.2. Shortcutting skips executing the standard entry and exit code. Which can add up to around a third of the total system call latency.[3]

The keen reader will be ready with several objections to the wisdom of intermeshing application and kernel code in this fashion. One set of challenges has to do with the difference between the application and kernel notions of stack-based memory. In shortcutting system call paths, we make sure to switch onto the kernel stack before executing the kernel handlers. Next, we outline some of the considerations we've noticed in our experience building UKL and realizing Dynamic Privilege, though ultimately, we ask the reader to hold these objections until Chapter 5, where we

[3]We have also experimented with calling kernel functions that are not system call handlers. One such example was allowing direct application-level polling on network queues. Instead of spending time learning existing OS interfaces or architecting them ourselves, we made use of direct calls to function like `napi_complete_done()` to signal the completion of management of a given queue from the kernel polling subsystem.

discuss stack issues in detail.

While the fundamental execution model in the kernel and user processes is consistent, involving the CPU fetching and executing instructions from memory, certain implementation properties differ. For instance, most contemporary software adopts a "stack-oriented" approach. This method utilizes a designated memory region to create a program stack, which compilers use to implement function calls through specific CPU instructions and registers. In typical UNIX operating systems, user processes are provided with extensive virtual memory regions for their stacks, leading to software development under the assumption of having large, dynamically-backed stack memory. Conversely, in environments like the Linux kernel, stacks are notably limited in size. The Linux kernel also eliminates stack page faulting by pre-allocating and pinning stack memory, and mapping every stack across every core. This avoids complex page faulting behavior, contrasting with the more flexible stack usage in user processes.

1.5 Using Kernel Types, Macros, and Inline Functions

The previous exemplars may be powerful low-level building blocks, but their functionality is far from the full, OS-level abstractions kernel developers need to work efficiently. The low-level hardware access we have demonstrated provides a foundation for the more robust OS-level access we explore in this section. We begin by demonstrating the ability to use OS-level abstractions like kernel types, macros, and inline functions to write new code for elevated processes. These tools and the dynamic linking of elevated processes with kernel code (see Section 4.1.6) allow for the practical reuse of kernel functionality in user code. This is how we build up from low-level hardware access[4] to rich OS-level access[5]. First, we provide some necessary

[4]Like accessing kernel memory.

[5]Like Linux's architecture agnostic interfaces to modifying virtual address spaces.

background before diving into the example. We discuss this background from the perspective of the Linux kernel, but it will apply to other C kernels and, likely, most complex compiled software projects.

Background

The source code of the Linux kernel is a human-readable representation of a complex system. It utilizes features of the C language extensively to balance performance, readability, and maintainability, among other concerns. Key among these features are macros, inline functions, and complex type definitions. These type definitions, such as complex data structures, emulate features of object-oriented languages, akin to C++'s polymorphism. These "compile-time" phenomena are essential for developer productivity, yet they typically leave little or no runtime footprint. This creates a challenge for elevated processes that aim to utilize the full range of kernel abstractions. Macros, inline functions, and complex types are not merely infrequent corner cases to be addressed separately; they are ubiquitous throughout the kernel. Therefore, a systematic solution is required if Dynamic Privilege is to provide OS-level access to the host system effectively.

Macros: Preprocessor macros are a form of metaprogramming (code writing code) where the preprocessor (the first stage of compilation) is used to generate C code for the compiler to consume. Preprocessor macros are essential for Linux kernel code generation, compile-time optimization, conditional compilation, and architecture-independent abstraction. An informal search of the Linux source code .c and .h files turns up 5.3 million instances of the string `#define` (note this is counting instances across multiple architectures), the definition of a macro.[6] This extensive use of macros is almost closer to building a project-specific programming language rather

[6] Accomplished running the following command in the top-level Linux directory: `ag --cc --hh --nocolor --nofilename --nobreak ‘‘#define’’ | wc -l`.

than merely an occasional programming convenience.

One use of macros occurs when a source file uses `#include` to expand a header within the compilation unit. Macros are also a way to encode constants that, while necessary at compile time, do not need to appear in (e.g.) the data section of the kernel binary (where they risk polluting data caches at runtime). Macros serve as the basis for the kernel's Kconfig system that conditionally defines system components to appear in the compilation with `#ifdef`. The kernel is notorious for deeply recursively-nested macros like those to implement its linked lists (e.g. see `LIST_HEAD`). Macros are also used to abstract over architecture-specific implementation, like the `pgd_offset` and `pmd_offset` macros that allow navigation through the page table hierarchy across multiple processor architectures. We actually use these in elevated code, producing application code that works across architectures with an application re-link.

Inline Functions: Inline functions are distinct from the perhaps more familiar "standalone" functions like `ksys_write` which we executed in the previous section, Section 4.1.4. The compiler generates proper assembly code for standalone functions that follow the C calling conventions, which dictate how parameters are passed and how return values are received. Generating code that can execute standalone functions from any language is easy; they are instantiated once, and importantly, we can retrieve the kernel virtual address of standalone functions from the kernel symbol table (or the kallsyms subsystem).

By contrast, inline functions meet none of the above properties. When the compiler "in-lines" a function, it expands the function call at the call site within a standalone function. This may happen many times (everywhere it is called), it does not follow the C calling conventions, and it doesn't have a meaningful virtual address. Inline functions are mainly useful for performance optimization. Inlining allows inter-procedural optimization between functions (co-optimization), such as: eliminating stack frame

setup, constant propagation, loop unrolling, and hardware optimizations like branch prediction and register allocation optimization. One concern with inlining is code bloat, where the binary can grow (with multiple implementations of the same logic), causing instruction cache performance to suffer. An informal count of the appearances of "inline" in .c and .h files turns up 97 thousand instances.[7]

Much basic functionality is implemented in the Linux kernel as inline functions. One example is `__to_fd()`, which translates a process's file descriptor (an integer type) into the corresponding kernel fd struct, the functionality we needed when implementing deep shortcuts in Section 4.2.2.

Complex Types: The Linux kernel utilizes highly complex variable types. The Linux kernel even packages a subsystem for its e-BPF system to access better function and type information, the BTF Type Format (BTF) subsystem. In C, structures are types. The `task_struct` definition, as an example of a complicated type, spans over 1,400 lines of code, many components of which are complex types themselves. Finally, while C does not offer direct support for runtime polymorphism like an object-oriented language, the Linux kernel still wants to exploit this degree of freedom when, for example, loading device drivers at runtime. To provide this, the kernel uses structures that resemble the Virtual Tables (Vtables) of function pointers, such as the `file_operations` struct as seen in Figure 4·7 Writing elevated application code to allocate, index, and traverse these types requires significant support. For example, when calling an internal kernel interface, elevated processes will often need to synthesize arguments of complex types, like the `struct msghdr` we used when making the deep shortcutted call to `tcp_sendmsg`.

[7]Executing the following search command: `ag --cc --hh --nocolor --nofilename --nobreak ''inline'' | wc -l`

```
1852  struct file_operations {
1853          struct module *owner;
1854          loff_t (*llseek) (struct file *, loff_t, int);
1855          ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1856          ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1857          ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1858          ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1859          int (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
1860                        unsigned int flags);
1861          int (*iterate_shared) (struct file *, struct dir_context *);
1862          __poll_t (*poll) (struct file *, struct poll_table_struct *);
1863          long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1864          long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1865          int (*mmap) (struct file *, struct vm_area_struct *);
1866          unsigned long mmap_supported_flags;
1867          int (*open) (struct inode *, struct file *);
1868          int (*flush) (struct file *, fl_owner_t id);
```

Figure 4·7: The `file_operations` structure enables runtime polymorphism across various devices. It serves as a function pointer table, with each pointer linked to a specific file operation (like open, read, write, or close). Device drivers fill this structure with their unique function implementations, allowing the kernel to interact with a wide array of devices uniformly.

Dynamic Privilege Properties Exemplified

This example shows that all the familiar kernel types, macros, data, and inline functions are available for use by elevated processes. The same high-level kernel abstractions available to kernel developers and module authors are also available to elevated processes. This example demonstrates properties 5b, 6, 7, 8, and 9 from Figure 4.1.

The above example, Figure 4·8, looks like standard kernel or kernel module code. Surprisingly, we build it into a shared object using a standard toolchain and dynamically link it into a privileged Python process. This short example demonstrates the use of kernel types, macros, and inline functions. "current" is a kernel macro that expands to a function call, `get_current()`, returning a pointer to the currently running task (thread). The arrow operator used when accessing the process ID (PID) data element in `current->pid` demonstrates that our tools understand the complex `task_struct` type, confirmed when the PID prints.

While `printk()` is a standalone function, which we have already shown we can execute, `fdget()` is an inline function. However, in application source code that builds elevated processes, we can utilize kernel inline functions by employing kernel head-

```c
static void fd_info(int fd) {

    struct fd f = fdget(fd);

    if (f.file) {
        printk(KERN_INFO "For pid %d, FD %d f_pos: %lld\n",
                                    current->pid, fd, f.file->f_pos);
    }

}
```

Figure 4·8: This C example from a elevated process is also valid Linux kernel module code. It demonstrates properties 6) kernel data access when recovering the PID value from the task struct; 7) the understanding of kernel types in being able to calculate the offset of the pid field within the task struct; 8) expanding preprocessor macros in resolving "current" the macro into the corresponding kernel function; 5a) in calling `printk()`; 5b) in the usage of `fdget()`, a static inline function which resolved into its implementation from kernel headers; 9) dynamic linking to a standalone kernel function `printk()`.

ers with the preprocessor, which allows for the expansion of these inline functions. This example demonstrates that we can use kernel types, macros, data, and inline functions from elevated processes.

Finally, taking a closer look at the usage of `printk()`. When we first started executing kernel functions, we would resolve their addresses at runtime using `kallsyms_lookup_name()`, the example in Section 4.1.4, using `ksys_write` from Python. This is a powerful approach because it allows runtime lookup of standalone functions, and it is easy to synthesize calls to many of them from the elevated user context. We have also explored a dynamic linking approach, the topic of the next section, Section 4.1.6.

The Larger Picture

"Standalone" kernel functions differentiate from "inline" functions because they follow calling conventions and have a single entry point at a known kernel address. In

contrast, inline functions are expanded and enmeshed into the calling function's body by the compiler. This expansion means that inline functions do not have a single entry point and are not callable from outside the kernel. We discuss handling inline functions in section 4.1.5.

Dynamic Privilege directly provides low-level hardware access. This example shows that it is possible to build up from that foundation to use the full range of kernel abstractions. This step is significant toward wedding OS-level access with the full-featured application programming environment.

1.6 Dynamic Linking

Unikernels, or library operating systems, typically statically link applications to system service libraries. UKL pioneered statically linking a process into the Linux kernel. This extended the application interface past the typical system call API, allowing the optimized process to exploit the internal kernel interfaces, including all kernel symbols, e.g. by creating custom entry points to the kernel. Using this, UKL specialized the system to the needs of the application, improving performance.

We explore a different tack. We are building towards the ability to *dynamically* link applications with the Linux kernel. We believe this is the first attempt to make a running kernel appear to an application (with access to all of its symbols) just as dynamic linking gives an application access to a user-level library's symbols. We investigated two primary approaches: building dynamically linked application executables and building dynamically linked shared objects ready for use by an elevated process. Our approach has limitations, such as managing kernel assembly code not built with position independence in mind. Nevertheless, we believe our method is sound, even though it is incomplete at this stage. This section progresses with four subsections: 1) a conceptual overview of application-level dynamic linking; 2) an overview of our approach to dynamically linking elevated processes to the Linux ker-

nel; 3) two examples, of creating an executable and a shared object; 4) a discussion.

Dynamic Linking Conceptual Overview

Dynamic linking, facilitated by tools like ld.so in Linux, connects application code to external libraries at runtime. Dynamic linking contrasts with static linking, where all of an application's libraries are copied into the application's address space at compile time. Dynamic linking allows for memory sharing of libraries between processes. The dynamic linker reads the application's binary to identify references to symbols defined in external libraries. For example, if an application needs printf.o from libc and sin.o from libm, the linker loads these from their respective files and links them, following POSIX dynamic linking standards. Linking updates the binary's data structures to address symbol references between objects.

Position Independent Code (PIC) is an important aspect of dynamic linking, enabling code to be relocated (placed anywhere in the application's virtual address space) in memory at runtime. This feature is commonly utilized with shared libraries, allowing them to be loaded at different addresses in different programs without conflicts.

Dynamic linking's late binding feature allows runtime symbol binding, simplifying library updates without recompiling application executables. It also enables symbol *interposition*, where additional objects can be loaded to override default symbol definitions. Interposition forms the backbone of our approach to transparently shortcutting applications. In Linux, this is often achieved using the LD_PRELOAD environment variable. For further details on dynamic linking and interposition, readers may refer to standard texts in the field, e.g. (Bryant and O'Hallaron, 2016).

Overview of Our Approach to Dynamically Linking and Elevated Process with the Linux Kernel

Our approach to dynamically linking elevated applications to the Linux kernel rides on one core observation: when an application enters hardware privilege, the running Linux kernel becomes accessible in the upper half of the application's address space. We consider this analogous to the way a shared library becomes accessible to an application when the dynamic linker loads it into an application's address space. All we need to do is educate the application on the location of the kernel's symbols.

In the dynamic linking context, we base our methodology on several key assumptions about the structure and compilation of code. In our approach, we divide code into two distinct components. The first component is the application source, such as app.c, which includes standard user headers for user libraries like stdio.h and math.h, along with an interface header for the Dynamic Privileged code, e.g., mydynpriv.h. The second component is the Dynamic Privileged source, a new component of the application code that uses Dynamic Privilege to exploit kernel functionality. The Dynamic Privilege component will exploit OS source code and symbols. This segment, represented by mydynpriv.c, exports a functional interface to the rest of the application (the rest of the application, app.c, is devoid of kernel source dependencies) with functions like `int mydynpriv_foo(int pid)` and `int shortcut_tcp_sendmsg()`.

Compilation Proper: Here, we consider just the compilation component of building binaries that dynamically link with the kernel; linking will be handled later. The compilation process for the application source components adheres to the standard toolchain settings, resulting in dynamically linked shared objects conforming to Linux's default configuration. However, the Dynamic Privileged source is constructed using a slightly modified version of the existing kernel module build system to ensure compatibility with the default user object configurations. This modification includes

using the user memory model on x86 systems, disabling the kernel's mc-model-kernel,[8] special-casing per-CPU code,[9] and circumventing custom assembly code that assumes kernel-specific address spaces. Crucially, it also entails compiling with position independence `-fPIC` to enable dynamic loading and linking of the new code with standard relocation techniques.

We explore two primary methodologies for linking. The first method is a restricted approach that involves using a custom linker script, which extends the default application linker script with kernel symbols required by mydynpriv.c, meeting the specific requirements of the application. This provides the application with the virtual address locations of kernel symbols like standalone functions and data.

The second method is creating a more flexible and complicated shared library that describes the kernel symbols, libkernel.so. This approach involves building libkernel.so as a standard user dynamically linked library, starting from /proc/kallsyms, which lists all the symbols of the running kernel.[10] The resulting libkernel.so contains a symbol table but no loadable sections. Initially, all symbols are registered as "absolute" (implying that they have a fixed address in memory), but the goal is to match these with the corresponding kernel binary sections in the future. While symbols are absolute, we want to support the full flexibility of relocatable symbols, supporting the runtime update of symbol addresses relative to their section bases. This process involves using objcopy to add symbols to an empty libkernel.so and then simply adding -lkernel to the application link line. Examining the binary's symbol table shows kernel symbol references are undefined, and appropriate Procedure Linkage Tabels PLTs

[8]The kernel memory model builds code that is only compatible with pointer references to addresses in the upper 2G of the 64-bit address space, which is not where Linux applications execute.

[9]On x86, this code exploits segment base register relative addressing which does not directly integrate with position independence.

[10]Kallsyms is a powerful source of symbols because it stays current with all kernel symbols, including updating with the insertion of modules, etc. Contrast this with the kernel ELF, vmlinux, which would not provide the correct symbol addresses after kernel address space layout randomization (KASLR), for example.

and Global Offset Tables GOTs are created. LD_DEBUG (printing debug information when loading the executable) confirms that libkernel.so is actively utilized at runtime. Despite the symbols being absolute and their flags unset, their dynamic relocation in the disassembly and the fact that they are undefined in the executable is strong evidence that dynamic relocation functions as expected.

As acknowledged above, our current approach to dynamically linking an elevated application with the kernel suffers from limitations primarily due to issues of position independence. Segment register-based addressing, and custom assembly code are two current corner cases we hope to fully support in the future. If these constraints are avoided, a generic approach to dynamic linking is feasible. However, if these constraints are not met, alternative solutions or adjustments in requirements must be considered, such as detecting violations and resolving them with alternative code or build procedures.

Two Examples

Building a Dynamically Linked Executable Prepare a kernel module source file (e.g., `kmod.c` of Figure 4·9). Compile the module as a normal Linux kernel out-of-tree module using a make target (see Figure 4·10). The focus should be on the intermediate `.o` object file produced, which can be linked into a normal userspace application. Functions defined in this file can be called post-elevation and execution with privilege. During linking, a helper linker script resolves undefined kernel symbols (see Figure 4·12). Finally we share our final link line (see Figure 4·12).

```c
#include <linux/module.h>    /* Needed by all modules */
#include <linux/kernel.h>    /* Needed for KERN_INFO */
#include <linux/printk.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anonymous Template");
MODULE_DESCRIPTION("kmod");
MODULE_VERSION("1.0");

void __kmod_kprint(const char* msg) {
    printk(msg);
}

int init_module(void) {
    return 0;
}

void cleanup_module(void) {
}
```

Figure 4·9: Kernel interfac, kmod.c.

```
kmod: kmod.c
	make -C $(LINUX_PATH) M=$(PWD) modules
```

Figure 4·10: Utilizing the kernel module out-of-tree build system to create kmod.o.

```
INCLUDE default_ld_script.ld

printk = 0xffffffff81c76c1d;
pgdir_shift = 0xffffffff826f152c;
boot_cpu_data = 0xffffffff82b63640;
physical_mask = 0xffffffff82708310;
sme_me_mask = 0xffffffff8284def8;
ptrs_per_p4d = 0xffffffff826f1528;
pv_ops = 0xffffffff8283cc40;
page_offset_base = 0xffffffff826f1520;
phys_base = 0xffffffff8281a010;
find_get_pid = 0xffffffff811085e0;
put_pid = 0xffffffff81108490;
pid_task = 0xffffffff811082b0;
current_task = 0x0000000000017bc0;
```

Figure 4·11: Trivially extending the default application linker script (obtained from ld) to include the necessary symbol definitions.

```
main: main.o kmod.o
	$(CC) $(CFLAGS) -o $@ $^ -lSym -L$(SYMLIB_DYNAM_BUILD_DIR) -T symhelper.ld
```

Figure 4·12: Building executable.

Building a Dynamically Linked Shared Object Due to shared libraries on Linux requiring position-independent compilation (using the `-fPIC` flag), direct linking of the kernel module object file is not feasible. The kernel's memory model, which prohibits using the `-fPIC` flag, is incompatible with position-independent code. The compilation process may be broken down as follows. First, resolve includes and struct definitions with the GCC `-E` flag, producing a preprocessed source file. Next, compile the preprocessed source into an assembly file, omitting flags such as `-I`, `-include`, `-nostdinc`, `-isystem`, `-D`, and `-mcmodel=kernel`, and adding the `-fPIC` flag to create position-independent assembly (`kmod.s`). Finally, use the GNU assembler (GAS) to assemble the resulting file into a final object file for safe linking into a userspace shared object dynamic library.

Compilation Stages: `kmod.c` → `preprocessed_kmod.c` → `kmod.s` → `kmod.o`

Discussion Dynamic linking of elevated processes to the kernel provides several key benefits: robust access to kernel symbols, enabling OS-level access; runtime selection of elevated programs; parallel execution of multiple elevated processes; avoidance of challenges associated with static linking of kernel and application codebases.

Dynamic linking also allows for the creation of shared libraries containing elevated code, which can be used with the `LD_PRELOAD` environment variable to transparently insert elevated code into existing applications through system call interposition. This forms the basis for the transparent optimization approach discussed in Section 4.2.2.

1.7 Conclusion

This section demonstrated our approach to incrementally building up from the low-level hardware access that Dynamic Privilege provides directly, up to OS-level access. On the hardware side, we showed that elevated threads can access privileged instructions, registers, and hardware data structures. On the OS side, we showed elevated

threads can execute standalone kernel functions and exploit the various macros, inline functions, and complex types, which leave little runtime footprint, but are key to kernel development. Finally, we showed a limited example of dynamically linking elevated executables to the kernel and a more general, but incomplete method for dynamically linking elevated shared objects to the kernel. In summary, the chapter highlights integrating Dynamic Privilege access across various abstraction levels, while retaining the personality and rich user-level tooling of application ecosystems.

2 Application Optimization via Shortcutting

General-purpose operating systems implement significant functionality that applications can explicitly invoke via the system call interface. This functionality often involves interfacing with other processes on the system and devices managed by the OS.[11] These requests are frequently made during performance-critical tasks, such as writing to storage or networking devices, as part of completing a latency-sensitive computation. This makes them a target for performance optimization. This section discusses research approaches to optimizing system call paths and then explores how Dynamic Privilege can enable a particular form of system call optimization, called "shortcutting," as described in our previous work, UKL(Raza et al., 2023).

Using Dynamic Privilege via kElevate, applications can directly invoke the in-kernel entry point of system call handlers, avoiding context-switching into the kernel protection domain and common kernel "entry" code (see Section 4.2.2). This specialization can allow an application to balance context-switch overheads against the functionality it needs from the standard "entry" code by selectively invoking the "entry" code rather than executing it on every system call. We categorize shortcutting optimization into two types: "Shallow" vs. "Deep." The distinction lies in the depth and

[11] E.g., Interprocess communication or writing data to a storage or network device.

specialization of the kernel code that is used as the entry point. We demonstrate the effectiveness of shortcutting in improving throughput, latency, and energy efficiency across various operating contexts, such as virtualized and baremetal server environments. Our approach can employ dynamic linking and maintains full ABI compatibility, allowing existing dynamically linked binaries to be shortcutted and optimized without modification. We present both "shallow" and "deep" shortcutting techniques and quantify their impact on application performance through microbenchmarks and experiments shortcutting the Redis server.

Ultimately, we will demonstrate that specialization via shortcutting enables simultaneous optimization of application throughput, latency, and energy consumption. A microbenchmark that issues `write()` system calls in a tight loop to a RAM disk file sees a throughput improvement of 37%, a 54% reduction in 99% tail latency, and reduce total energy consumed by about 36%. An unmodified Redis server sees a 22% improvement in throughput.

This chapter is organized into three subsections. Section 4.2.1 provides background on syscall optimization and shortcutting. Section 4.2.2 describes our methodology, including our shortcut implementation and user interface to shortcutting. It also covers our experimental setup. Finally, Section 4.2 provides experimental evidence demonstrating the efficacy of our shortcutting technique.

2.1 Prior Work on System Call Optimization

The system call API is the primary interface between the application and the kernel. Access to performance-critical devices like storage devices and NICs is done through issuing system calls on application critical paths. In addition to the "direct" latency cost of issuing syscalls, there is an "indirect" cost that the application pays in reduced processor efficiency, a lower instructions per cycle (IPC) due to microarchitectural effects like cache polution(Soares and Stumm, 2010). As a result, reducing

system call overhead has been a focus of significant research and engineering efforts. Approaches range widely in terms of how much they demand in terms of porting and how much optimization is actually achieved. This subsection reviews two common approaches that have been explored: batching and the elimination of hardware overhead for system call optimization. Section 4.2.2 describes our alternative approach, "shortcutting," in detail.

Batching In the context of Linux, mechanisms such as 1) FlexSC's binary-compatible exceptionless syscalls (Soares and Stumm, 2010), and 2) Linux's `io_uring`(Arch Linux, 2020) use asynchronous issue and batching to reduce the overhead of system calls. `io_uring` is relatively simple; it allows applications to submit I/O requests to the kernel without system calls, reducing context-switch overhead. Applications request the creation of submission and completion queues shared with the kernel. The application submits I/O requests to the submission queue, and the kernel processes them asynchronously, placing results in the completion queue. `io_uring` supports various I/O operations, request batching, and chaining, enabling high-performance, low-latency I/O for I/O-intensive workloads. `io_uring` requires porting effort by modifying application source code.

FlexSC(Soares and Stumm, 2010) offers a specialized binary-compatible interface by providing a user-mode thread package (POSIX thread compliant) that can automatically batch system calls for applications that maintain large numbers of application-level threads. In this M-on-N model (M user-level threads are mapped to N kernel worker threads and M much greater than N), FlexSC utilizes the independence of user-level threads to optimize system call execution. FlexSC will queue one thread's syscall request, switch to another thread awaiting its next system call, queue it, and so on. Once a certain number of syscalls are queued, the system can make a single context-switch onto a kernel thread to execute the batched syscalls, reducing the num-

ber of context-switches required. Further, the kernel scheduler can isolate syscall work on a separate core from the application to improve caching for both. FlexSC assumes that mutual exclusion is handled at the application level, allowing the user-mode thread package to manage synchronization between the user-level threads. This assumption simplifies the implementation of the user-mode thread package and enables more efficient batching of syscalls without concerns about potential race conditions or conflicts.

Eliminating HW Overhead The hardware component of system call costs has seen significant industrial optimization. In the x86 case, `int 0x80` was improved upon by `sysenter` and eventually the `syscall` instruction. Despite this effort, transitions are 100 times more expensive than a function call instruction, even when using modern instructions. Kernel Mode Linux (KML)(Maeda and Yonezawa, 2003) is a Linux kernel patch that runs user executables in kernel privilege to replace the expensive x86 `int 0x80` system call exception with simple function calls. Given that the entire application is running with kernel privilege, KML can replace all system call instructions occurrences with function calls to the kernel's common system call handler function. This is attractive because it requires no changes to the user-level application. Applications can run without modification if isolation is not required. When isolation is needed, the authors suggest using software techniques like static type checking and fault isolation, as exemplified by their use of Typed Assembly Language. Work building on KML, such as Lupine(Kuo et al., 2020a) and Privbox(Kuznetsov and Morrison, 2022a), introduces configuration-time specialization and privileged compartments, respectively.

X-Containers (Shen et al., 2019) employ an exokernel-inspired architecture, where XEN (Barham et al., 2003) partitions provide strong isolation between containers. Within X-Containers, user-mode application processes are weakly isolated. Inside

an X-Container, Library OSes (X-LibOS) run in user mode, enabling function calls to replace most system calls. This approach avoids the overhead of virtualization incurred by models like Dune (Belay et al., 2012) (see Section 2.2.2). However, the trade-off is that isolation between processes within a container is removed, and there is no protection between processes and the libOS. The benefit of this design is significantly reduced context-switch costs between user and kernel mode within a container, as system calls (and the associated hardware overheads) are replaced with function calls.

The next subsection addresses our approach to optimizing the system call path via "shortcutting." Shortcutting eliminates the hardware portion of the context-switch as well as enabling the selective by-passing of kernel software costs. UKL demonstrated that the software component of this cost is critical to fully reap the benefits of system call optimization.

2.2 Shortcutting Approach

Shortcutting, as introduced by our previous work Unikernel Linux (see Section 2.1.2), takes a different approach to reducing system call overhead, as compared to prior work described in Section 4.2.1. Shortcutting involves bypassing kernel software in addition to eliminating the overheads associated with hardware context and privilege switching on the system call path. This is done by directly invoking kernel syscall handlers or even deeper components of a syscall path, as we will describe. General-purpose operating systems perform non-trivial work on system call entry and exit paths. These transition regions are convenient places for the kernel to update its bookkeeping and perform maintenance. For example, the kernel will use this point to synchronize the Read-Copy-Update (RCU) subsystem, check if the scheduler should be invoked, and check for pending signals. While the kernel work that gets executed on system call paths should not be starved, our work on UKL found that many

applications can tolerate a significant reduction in the frequency of executing these paths (e.g., only 1 out of 20 times). That said, delaying this work too long can increase tail latencies, which may not be suitable for some applications. Therefore, shortcutting is a form of specialization that may be appropriate for some applications and not others.

Prior Work: UKL Shortcutting

Section 2.1.2 has an in-depth treatment of the UKL approach to shortcutting. In that work, we demonstrated that the hardware cost of the system call path (on x86_64) (the `syscall/sysret` pair) was small (e.g., 1-5%) of the total system call latency, whereas eliminating both the hardware and software costs via shortcutting lead to much more significant savings, between 37% and 83%. However, there are some limitations to UKL's shortcutting, which we address using Dynamic Privilege. The UKL approach requires a significant architectural reorganization. UKL demands the a priori selection of a single process to optimize. The binary for this process must be statically linked to the Linux kernel. Furthermore, the code of the application needs to be re-compiled with 1) flags to match the kernel, 2) built as a static executable, and 3) linked to a custom glibc. Given UKL's architectural requirements, shortcutting optimization is inherently limited to conforming use cases. For example, since UKL does not support `fork()` for the primary application, one cannot optimize those that do.

Dynamic Privilege Shortcutting

The Dynamic Privilege approach to shortcutting demonstrates that it is possible to enact shortcutting without the major architectural changes that UKL requires. One approach employs dynamic linking interposition and thus can maintain ABI compatibility with the existing application. This approach allows one to explore

shortcut optimizations without changes to the application binary or build system (see figure 4·13 and listing 1.1). This applies both to what we call "shallow" and "deep" shortcuts (see Section 4.2.2). There is also the option to do static linking, we explore both in the coming microbenchmarksSection 4.2.3.

We developed a shortcutting shell script (see Section 4.2.2) designed to be flexible, allowing the user to specify a list of functions to elevate or lower around, as well as a list of functions to shortcut. The tool relies on the executable being dynamically linked with its C library to perform the necessary interposition.[12] Our approach to shortcutting addresses a number of limitations of our prior work on UKL. We will show that this approach enables shortcutting of standard dynamically linked applications without any source code changes, without modification to glibc, and without any changes to the application build system. Further, the user can freely select which process and how many they want to shortcut at runtime, and applications can `fork()`. Finally, shortcutted apps can be deployed using standard command line tools. This makes it natural to deploy optimized applications through standard scripting and use software conveniences like containerization.

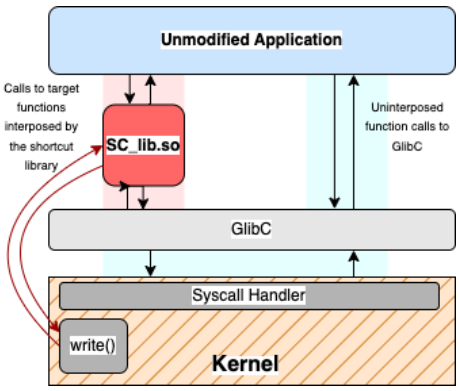

Figure 4·13: Transparent system call shortcutting using dynamic library interposition. Function calls intended for a C library like glibc are interposed. The SC_lib decides if it will forward the request to glibc or directly invoke the kernel handler.

[12]We have shortcutted statically linked applications, though our approach required a re-link.

Listing 4.1: A program, wrLoop, executed: 1) as a standard Linux process, 2) shallow shortcutting all write syscalls, 3) deep shortcutting *only* the single write call at text address 0x42ed60. In practice, more `-s` options can be added as a more sophisticated application is optimized. While this example is complete, there are many other options we don't demonstrate here.

```
$ ./wrLoop $SZ $ITER $PATH
$ ./sc.sh -s write->__x64_sys_write \
    --- ./wrLoop $SZ $ITER $PATH
$ ./sc.sh -s \
    write->tcp_sendmsg:0x42ed60 \
    --- ./wrLoop $SZ $ITER $PATH
```

Shallow Syscall Shortcutting Our shallow shortcutting of system calls is straightforward, as shown in Figure 4·13. We dynamically load our shortcutting library `SC_lib` ahead of the C library, hooking the specified syscalls. This library can either forward calls to the C library as the application intended or directly call the corresponding system call handler, leveraging kElevate's access to kernel code paths (see the blue arrows in Figure 4·14). In this way, C library function calls like `read()` (which would normally result in system calls to the OS) instead become function calls to the corresponding system call handler. This skips over both the `syscall` instruction and the kernel transition software described in Section 4.2.2.

Deep Syscall Shortcutting Deep shortcutting is a more specialized technique than shallow shortcutting, and can achieve greater performance improvements. A `write()` syscall can be destined for a normal file or any type of device, such as a network or storage device. The destination is unknown until after kernel disambiguation logic at the virtual filesystem (VFS) layer. Deep shortcuts make assumptions about where a particular write is going and short-circuit the kernel's routing logic. For example, consider Figure 4·14. This is part of a flamegraph(Gregg, 2016) produced by

periodically sampling the Redis server running on Linux. Redis makes `read()` and `write()` function calls into glibc, which makes system calls into the kernel. While the Redis server contains 29 unique calls to glibc's `write()` function, a single one is on the critical path to the network device. By using `perf` to determine the address of the write syscalls on the critical path one can selectively deep shortcut these. As demonstrated in Section 4.2.3, exploiting this "deep" shortcut opportunity in Redis results in significant improvements, e.g., 21% higher throughput on a baremetal server.

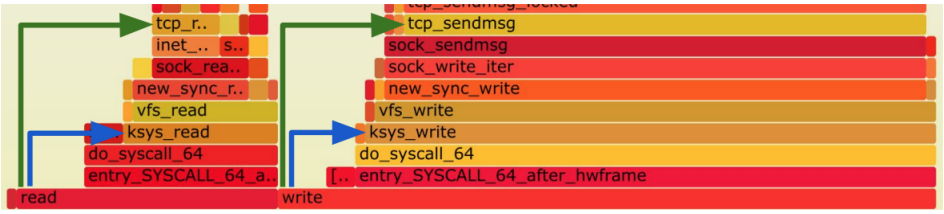

Figure 4·14: Part of a flamegraph generated from periodic sampling of a Redis server. Bar width is proportional to the number of samples taken. Collated stacked bars show a stack trace at an instant in time. Running on Linux, Redis makes function calls into glibc's `read()` and `write()` functions. Blue arrows show shallow shortcuts into the generic system call handlers. Green shortcuts show deep shortcuts into `tcp_sendmsg` and `tcp_recvmsg`.

As described next, this is done without annotation or source modification to maintain ABI compatibility. Using the tools and mechanisms discussed, the user can specify to shortcut only a particular invocation of a function by providing its address of the call site in the application binary, which can be determined using the symbol table if provided, or external monitoring tools like `perf` or `uprobes` through tracing if not. This approach allows users to minimize their perturbation to the system and provides the option to shortcut multiple `writes` on different critical paths to other devices.

Shortcutting Tools

We built two main tools to implement convenient application shortcutting from the command line. The first is the shortcut library, `SC_lib`. The second is a shell script that provides a simple interface for utilizing the shortcut library, `shortcut.sh`.

`SC_lib` The shortcut library, `SC_lib` serves as the core of our approach to shortcutting. The `SC_lib` is controlled by a number of environment variables and signals. The library defines a macro that can be used to interpose on any C library function that maps to a Linux system call. When the target process has started executing, the library *hooks* target functions provided to it as options. These functions are the C library wrappers for system calls such as `write()` On first calling a hooked function, the library determines and caches the address of the corresponding system call handler using the `kallsyms_lookup_name()` function (see Section 4.1.5). On every call, the `SC_lib` decides whether to forward the call to the C library or, instead, to exploit the access kElevate provides to call directly into the corresponding system call handler. When shortcutting, the library makes a standard syscall for every 20 shortcut calls[13] to reduce the chance of starving these system maintenance paths.

Implementing interposition as a macro is possible because (in the case of Linux running on an x86 processor), all system call handlers correspond to kernel aliases of the form `__x64_sys_<syscall_name>`, e.g., the `write()` system call corresponds to the kernel symbol `__x64_sys_write`. While system calls can take variable numbers of inputs, all of these `__x64_sys_<syscall_name>` functions take only a single argument, a pointer to a struct containing the application register state when the call was made (so arguments can be read directly out of the registers according to the calling conventions, e.g., `rdi`, `rsi`, `rdi` ...).

The shortcut library also uses environment variable flags to determine if the application will start up running with or without privilege. Because this is handled in a constructor that runs before the application begins, the entire application will run in this mode. Finally, the library is designed to utilize runtime signals. These can be used to toggle elevation status or shortcutting on and off at runtime. The latter is

[13]This is an ad-hoc policy that we expect to be configurable and tuned as needed. E.g., in our microbenchmarks we set it to 10,000.

useful for observing the performance difference of an application with and without shortcutting.

The Shortcut Script, `shortcut.sh` `shortcut.sh` (see help menu in Figure 4·15) is a shell script used to prepare the necessary environment variables for preparing environment variables for the shortcut library and starting the target process. It uses the LD_PRELOAD trick (common for interposition) to load the `SC_lib` ahead of the C library, resolving any functions the library is responsible for interposing. Section 4.2.2 shows some sample usage of the shortcut script (abbreviated sc.sh). Command 1 is how the program `wrLoop` would be executed on the command line.

```
[kele@rfc1918 shortcut]$ ./shortcut.sh -h
###################################

shortcut.sh [args] --- <app> [app args]

Script for launching apps with options for elevation and shortcutting.

Flags:

        -p: passthrough mode, run application without interposer library.
                Ignores all other args,
The following options may interact with kElevate

        -be / -bl: begin with executable elevated / lowered

These ones  can be repeated to specify multiple functions:
        -e <fns>: Elevate around these functions
        -l <fns>: Lower around these functions
        -s <fn1->fn2>: Shortcut between these functions: eg: 'write->ksys_write'
                Don't forget to use quotes around the function names!

Debugging options

        -v: verbose mode for debugging
        -d: dry run, do everything EXCEPT running the application

###################################
```

Figure 4·15: The shortcut script, `shortcut.sh` help menu.

Command 2 uses the shortcut script to preload the `SC_lib` ahead of `wrLoop` and prepares environment variables. The `-be` flag sets an environment variable instructing the library to "begin elevated" by elevating in the constructor. The `-s`

Listing 4.2: wrLoop run: 1) as a standard Linux process, 2) shallow shorcutting all write syscalls, 3) deep shortcutting *only* the single write call at text address 0x42ed60. In practice, more `-s` options can be added as a more sophisticated application is optimized. While this example is complete, there are many other options we don't demonstrate here.

```
$ $SZ=1 $ITER=$((1<<22)) $F_PATH=/tmp/tgt.txt
$ ./wrLoop $SZ $ITER $F_PATH
$ ./sc.sh -be -s write->__x64_sys_write \
    --- ./wrLoop $SZ $ITER $F_PATH
$ ./sc.sh -be -s \
    write->tcp_sendmsg:0x42ed60 \
    --- ./wrLoop $SZ $ITER $F_PATH
```

`write->__x64_sys_write` flag results in setting an environment variable which instructs the library to perform shallow shortcutting, mapping all `write()` function calls to the corresponding handler, `__x64_sys_write`.

Command 3 enacts our deep shortcut. While it similarly preloads the `SC_lib` and starts the program running elevated, it does two things differently from the previous command. First, it does not apply to every `write()` function call, but instead only to the one at text address `0x42ed60`. Second, it maps the function call directly to `tcp_sendmsg`, instead of `__x64_sys_write`. This is implemented in the interposer code using a glibc built-in: `__builtin_return_address(0)`, which gets the address where the interposer will return to. If this matches what a user specified, we make the deep call. Otherwise, we have the option to execute it as a system call, or to shallow shortcut it.

2.3 Shortcutting Experiments

This subsection describes experiments we use to quantify the benefits of our approach to shortcutting using Dynamic Privilege. We start by describing the software and hardware environments we use (see Section 4.2.3). Then, provide microbenchmarks (see Section 4.2.3), and then consider shortcutting the networked key-value store

Redis as a macrobenchmark. We emphasize that this is not a first-class study of the optimization technique of shortcutting; for that, see the UKL paper (Raza et al., 2023).

Experimental Setup

We made a significant effort to control the software component of the experiments. Controlling software was eased by our approach to shortcutting, which preserves ABI compatibility, as the applications themselves required no modification. We carried out experiments on two hardware environments: a laptop and a pair of data center servers. All microbenchmarks are carried out on the baremetal laptop environment. Our virtual and baremetal Redis experiments are conducted on the server environment.

Software Configuration We conducted experiments on the Linux 5.14.0 kernel. The only software difference between our baseline Linux and the kElevate syscall is the use of the CONFIG_KELEVATE option in otherwise identical kernel configs.[14] We built the applications and kernel in a Fedora 35 environment (Fedora 36 cannot build the 5.14.0 kernel without compiler errors) and deployed and ran them on a Fedora 36 user-space environment, using the GCC and glibc versions as packaged for that distribution (glibc 2.35 and GCC 12.2.1 20221121 (Red Hat 12.2.1-4)). We used consistent kernel boot options, disabling SMEP and SMAP (see Section 3.2.2). We evaluated shortcutting on unmodified application binaries. Shortcutted applications are dynamically linked to our interposing shortcut library, `SC_lib` described in Section 4.2.2.

We test the Redis server in virtualization running in virtualization and baremetal. For virtualization, we used QEMU (QEMU, 2019) (version 6.2.0 (qemu-6.2.0-17.fc36)) with KVM (Contributors, 2019) with paravirtualized networking via virtio in all

[14]Items protected by this config are enumerated in Section 6.1.2.

virtual experiments. The UKL kernel (also based on 5.14.0) was built the same way, on a Fedora 35 environment and ran on a Fedora 36 user-space software distribution. We built Lupine and PrivBox according to the instructions found for the respective projects. We ran virtualized experiments on the "server" hardware (described next). As mentioned, we used two hardware environments for these experiments. We will refer to them as the "laptop" and "server" environments respectively. We utilize RAM-disk filesystems for all experiments. We ran experiments in single-user mode to reduce noise from daemons and other user-level processes. We use two hardware platforms described below.

Laptop This system is powered by an Intel Core i9-10885H processor, which has 16 logical cores (8 physical) and a base clock speed of 2.40GHz. The CPU can reach a maximum frequency of 5300MHz (5.3GHz) when boosted. It features 256 KiB of L1 cache (8 instances), 2 MiB of L2 cache (8 instances), and 16 MiB of shared L3 cache. The machine is equipped 62GB of RAM, providing ample memory for multitasking and running memory-intensive applications. Storage consists of two 953.9GB NVMe SSDs, with 8GB allocated for swap. This hardware was not used for networked experiments.

Server These twin systems are Supermicro servers, each equipped with two Intel Xeon Gold 6248 CPUs with 40 logical cores each (20 physical), a base clock speed of 2.5GHz and 768 GB of RAM split evenly between sockets. They have 6 SSDs disks, totaling 2.8TB of storage.[15] These nodes are connected for networking experiments by Intel 82599ES 10-Gigabit NICs. The system was configured with Hyper-Threading and Turbo Boost disabled and with the CPU governor set to performance.

[15]Two Intel SSDSC2KB24's at 240GB, two Intel SSDSC2KB96's at 960GB , and two INTEL SSDSCKKB24's at 240GB.

Shortcutting Microbenchmarks

Here, we quantify the cost of the kElevate mechanism itself and then study the difference between throughput, latency, and energy consumption. We will consider two programs here, `writeLoop`, a trivial loop making write system calls, and a modified version of LEBench microbenchmark(Ren et al., 2019) used in UKL(Raza et al., 2023). These microbenchmarks only study shallow shortcutting. `writeLoop` is a simple C "nanobenchmark," shortcutting this very simple program allows us to study the potential for benefits to throughput, latency, and energy. We will see that shortcutting can improve `writeLoop` throughput by about 37%, 99% tail latency by about 54%, and reduce total energy consumed by about 36%.

We demonstrate that our shortcutting approach generalizes to other system calls by shortcutting a collection of system call microbenchmarks derived from the LEBench(Ren et al., 2019) benchmark. Again, all of these microbenchmark experiments are carried out running baremetal on the "laptop" hardware configuration. Refer to Table 4.1 for details of the system configurations tested.

Term	Kernel	Privilege	Interposed	Shortcutted
Linux	Linux 5.14.0	X	X	X
kElevate (lowered)	kElevate 5.14.0	X	X	X
Interposed	kElevate 5.14.0	X	✓	X
kElevate (elevated)	kElevate 5.14.0	✓	✓	X
Shortcut	kElevate 5.14.0	✓	✓	✓
In-Source Shortcut	kElevate 5.14.0	✓	X	✓

Table 4.1: Summary of experimental configurations.

In the experiments of Table 4.1 "**Linux**" is the application running as a standard Linux process on the vanilla 5.14.0 kernel. "**kElevate (lowered)**" refers to the application running as a standard Linux process (lowered, or without hardware privilege), but on the modified kElevate kernel, which includes the (unused) kElevate system call. "**Interposed**" means the process is running lowered on the kElevate

kernel, and the shortcut library directs system calls to the standard glibc wrappers. "**kElevate (elevated)**" means that the process is running with hardware privilege on the kElevate kernel; the process is interposed but not shortcutted (system calls are going through the standard glibc wrapper functions). "**Shortcut**" means the process runs with hardware privilege on the kElevate kernel with privilege, and the dynamically linked interposing library directs system calls directly to kernel handlers. "**In-Source Shortcut**" means the application has been modified to call the system handlers directly, removing the cost of dynamic linking and interposition.

Questions Comparing between these configurations helps study some important questions. Comparing Linux and kElevate (lowered): "Did we change the performance profile of the kernel?" Our hypothesis is that there is no difference. Comparing kElevate (lowered) and Interposed: "Does interposition slow down the application?" We believe it should slow the application slightly as it adds an extra indirect jump on these performance sensitive paths and adds to the instruction caches. Comparing kElevate (lowered) with kElevate (elevated): "Does running application code with privilege affect the performance characteristics of the application?" We hypothesize that there will be no change. Linux and Shortcut: "Is ABI compatible shortcutting application shortcutting via dynamic linking a viable optimization strategy?" We hypothesize that it is. Finally, the shortcut and in-source shortcut are as follows: "What's the most we can get from shortcutting when we allow application modification?"

Microbenchmark Map The rest of the subsubsection provides quantitative microbenchmark results. In order, we will consider the base overhead of the kElevate mechanism; the `writeLoop` nanobenchmark including throughput, latency and energy; then LEBench derived microbenchmarks including a CPU bound loop, and the

following syscalls `getppid()`, `write()`, `read()`, `mmap()`, and `munmap()` syscalls.

Base Overhead of kElevate Mechanism To understand the overhead of the kElevate system call, we implemented a microbenchmark that consists of a loop performing back-to-back kElevate syscalls: first elevating and then lowering the privilege level. This elevate-lower pair takes 212ns, or 106ns per syscall, comparable to the average about 100ns cost of a `getppid()` syscall on the same machine, considered one of the lowest latency system calls. This latency bounds the granularity at which kElevate can be used on performance-sensitive paths.

While there is room for further optimizing the kElevate system call, e.g., by removing debugging code, we think extreme optimization can be handled orthogonally. For example, one might use Dynamic Privilege to create a fast path for the kElevate system call by interposing a specialization on the system call handling path. Ultimately, if kElevate were implemented in the hardware with a dedicated CPU instruction, we believe this cost could fall by at least an order of magnitude.

`writeLoop` **Nanobenchmark Results** Here, we study the baremetal execution of a simple C program that issues back-to-back synchronous one-byte `write` system calls to a RAM disk filesystem through the `glibc write()` wrapper. This experiment quantifies the benefits of applying shortcutting via kElevate to reduce the cost of the critical `write()` system call via shallow shortcutting (see section 4.2.2).

Figure 4·16 shows box plots comparing write throughputs for the various system configurations. These box plots are generated from 60 runs of the program. The Linux and kElevate (lowered) baselines differ by less than 3%, showing that the kernel changes alone didn't account for large differences in throughput achieved by shortcutting. We do note the apparent slight increase in throughput for the kElevate (lowered) case. We do not believe this is a reliable effect and are looking into whether

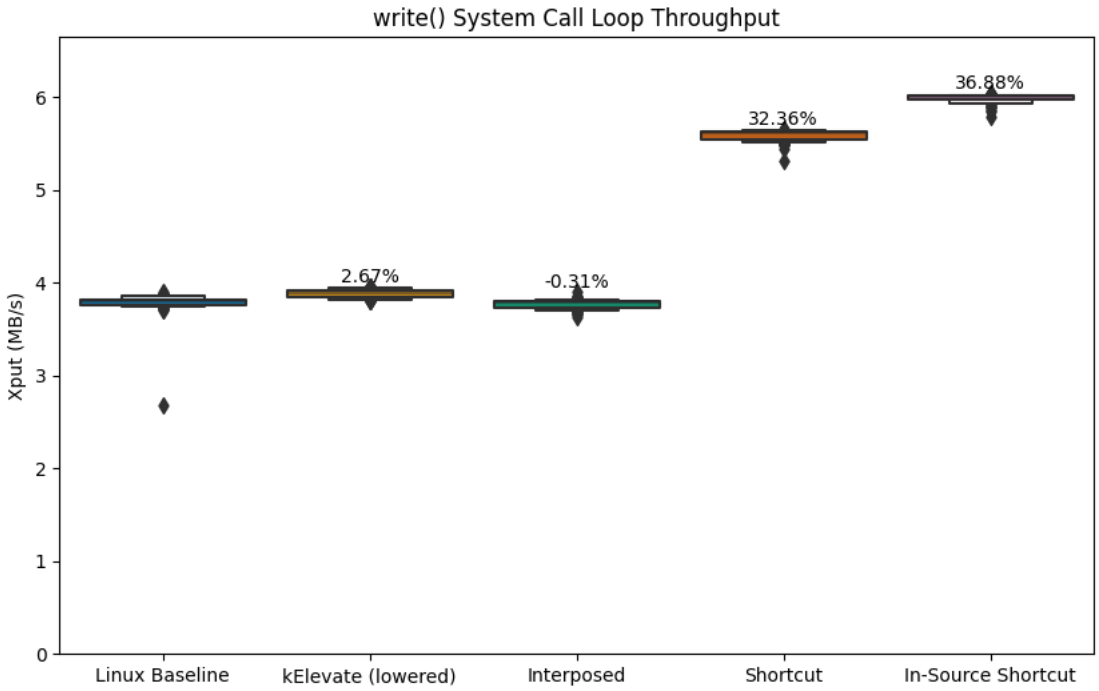

Figure 4·16: `writeLoop` throughputs measured in MB/s. Shortcutting provides a significant throughput advantage over the Linux and kElevate baselines. The In-Source shortcut configuration achieves nearly 37% higher throughput compared to the Linux baseline.

it fluctuates across system reboots. We flag this for discussion at the end of the microbenchmark section.

Figure 4·16 show a significant throughput advantage when using shortcutting. The shortcutted application using dynamic linking achieved 32% higher throughput than the Linux baseline. When modifying the application source code to eliminate the overhead of dynamic linking, we show a nearly 37% improvement in throughput.

The latency scatterplot of Figure 4·17 compares the latency of individually measured write system calls via the or shortcutted calls in issue order for the `writeLoop` program. We captured data across 60 runs. We select the 1 of those 60 that has the median throughput and plot the individual latencies for that run.

We can see the Linux baseline and kElevate (lowered) latencies roughly align into three bands, with the kElevate configuration performing slightly better in the upper

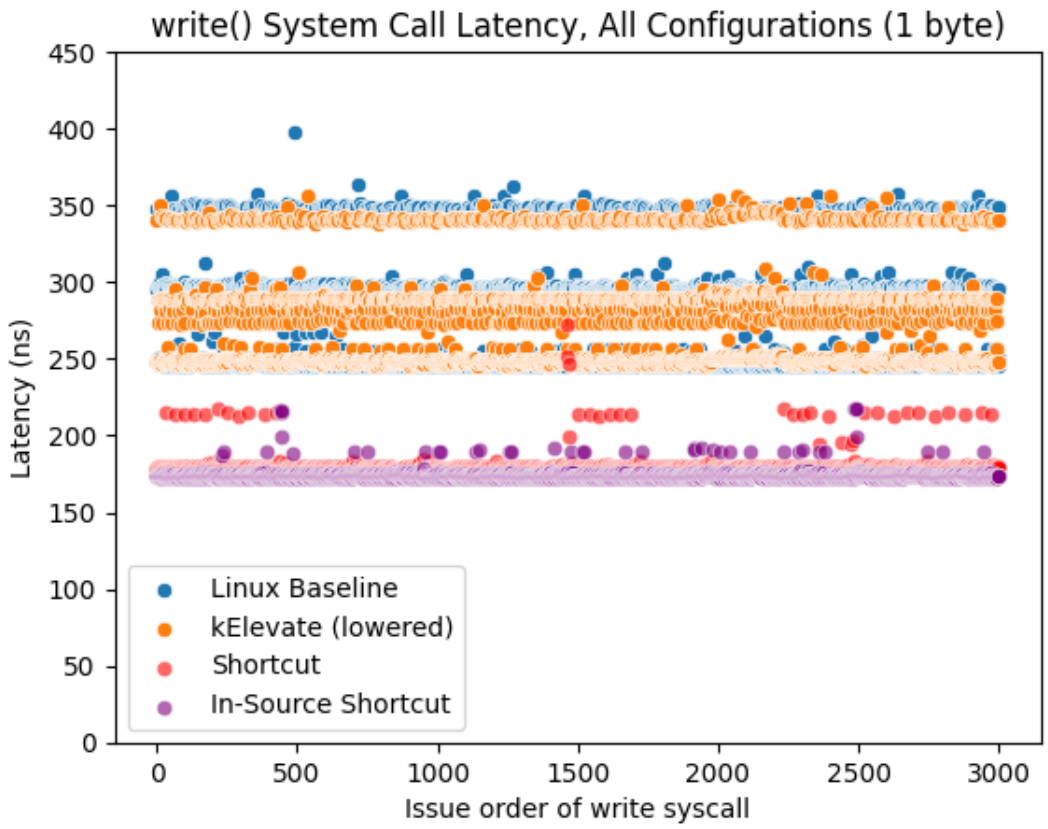

Figure 4·17: Scatterplot comparing the latency of individual write system calls or short-cutted calls. Shortcut and In-Source shortcut latencies are significantly lower than the Linux and kElevate configurations.

two bands. We do not know the exact cause for these multiple bands, even in the Linux case. We speculate that caching, or intermittent kernel work may be involved, but again, these are simple memory writes; they are not, e.g., destined for storage.

The shortcut and In-Source shortcut latencies roughly overlap at much lower latency values towards the bottom of the scattered points, with the In-Source Shortcut points at slightly lower values. While most of the data falls into one interesting point to observe here, it is that instead of three bands, we see one major band at a much lower latency.

Figure 4·18 summarize the latency data as three histograms. The three latency bands of the Linux baseline of Figure 4·17 appear as three clusters in the histogram. We

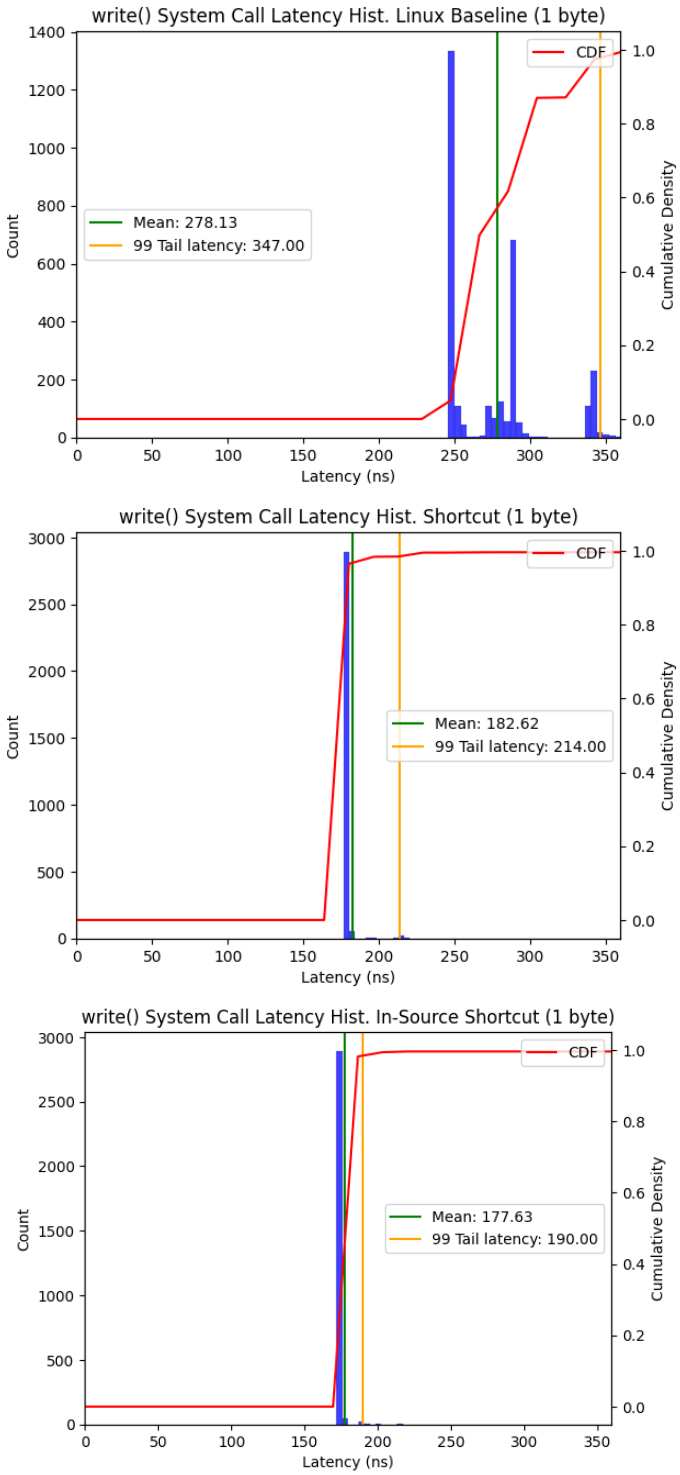

Figure 4·18: Latency histograms `writeLoop` for Linux, Shortcut, and In-Source Shortcut configurations. Notice the 54% reduction in the 99% tail latency, and 62% reduction in mean latency. Further, notice that the 99% tail latency of the shortcut cases is below the *minimum* of the Linux baseline. Note: Due to course bin-width discretization, the CDF line does not exactly track with the histogram bins.

emphasize the 54% reduction in 99% tail latency, 62% reduction in mean latency, and the fact that the 99% tail of the shortcutted cases is below the *minimum* of the Linux baseline.

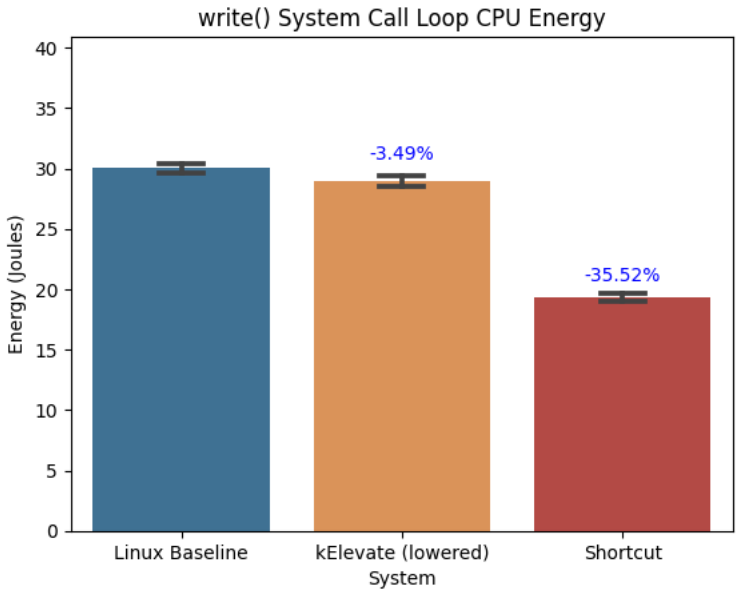

Figure 4·19: `writeLoop`: Energy consumed executing 2^{22} single-byte `write()` system calls. Measured across the "energy-cores" domain via Perf wrapping Intel RAPL. Intel's Running Average Power Limit (RAPL) energy reporting subsystem. Shortcutting results in a 36% reduction in Joules consumed.

Finally, we evaluate the impact shortcutting has on energy consumption of the `writeLoop` nanobenchmark. Figure 4·19 report the average (and standard deviation) amount of energy (measured in Joules) consumed by all cores and private caches on the laptop when `writeLoop` executes 2^{22} single-byte writes. This data was captured by Perf's "energy-cores" domain, backended by Intel's Running Average Power Limit (RAPL) energy reporting subsystem. While we see fairly comparable bars for the Linux baseline and k-elevate lowered configurations, the shortcutted case consumes almost 36% less energy for the same workload. We note the oddity of kElevate (lower) having a lower energy consumption, similar to the discrepancy in performance. We do not expect this to replicate.

LEBench Microbenchmarks Motivated by the findings of the `writeLoop` nano benchmark, we consider a broader set of syscalls to demonstrate that this intervention is generally effective, not just specific to tiny writes. We run LEBench (Ren et al., 2019) microbenchmark suite, designed to probe the performance of the core Linux system calls, slightly extended to reduce noise on memory allocation and to perform a more fine-grained sweep across different input sizes. We first calibrate and validate our methodology using the CPU-bound loop and `getpid()` tests of LEBench. Then, we examine a broader set of LEBench tests.

This is a single program that defines multiple system call microbenchmarks. Aside from the base CPU performance and `getppid()` boxplots (Figure 4·20 and Figure 4·21), We draw similar graphs showing latencies averaged across 30 runs of the microbenchmark. A single point on this graph is produced by averaging 30 values (one per run), each generated by averaging 10,000 individual system call latencies. The purple shading shows the maximum percent improvement between the Linux baseline and the fastest form of shortcutting. This is In-Source Shortcus for those that we implemented (`read()` and `write()`), and dynamically linked shortcuts for those we did not (`mmap()` and `munmap()`).

The box plots of Figure 4·20 are a measurement of how long it took to execute a CPU-bound loop of repeated floating-point division on the core. The similarity of these box plots is evidence that the processors were similarly configured at the hardware level, and thus the soundness of our methodology (please note the non-zero y-axis origin).

Figure 4·21 illustrates the impact of shortcutting the `getpid()` system call. `getpid()` returns a process's parent's process id. This is among the lowest latency syscalls; it simply looks up an entry in the process `task_struct`. Because it is so short, the constant latency shortcutting removed from the syscall path is the dominant cost of

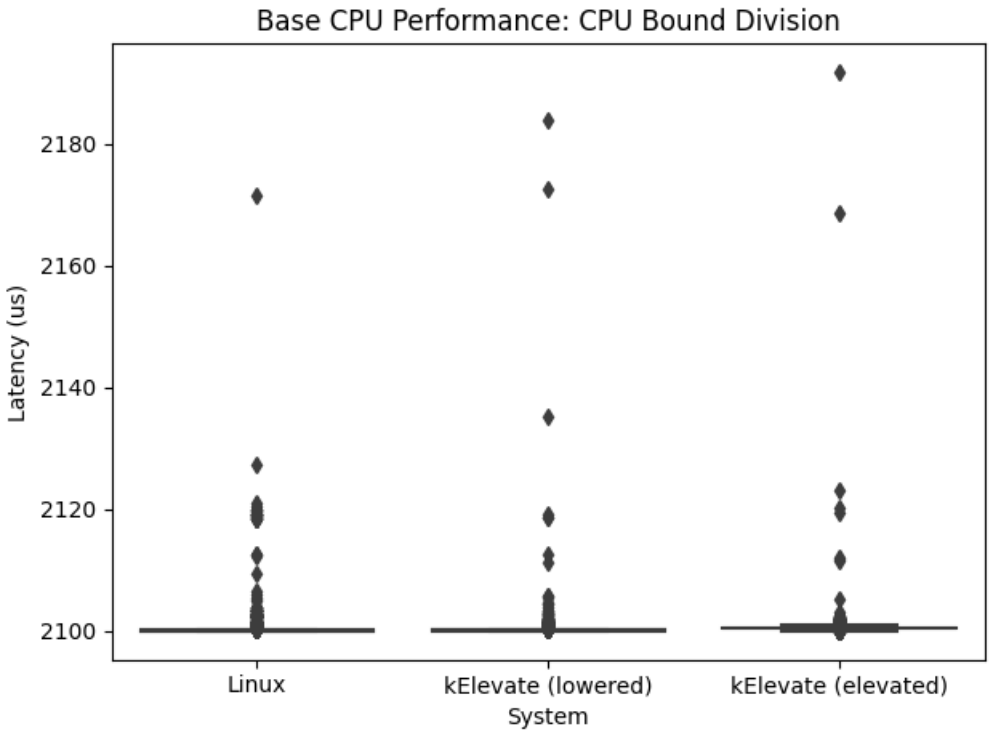

Figure 4·20: A CPU bound repeted floating point division loop. A calibration check orthogonal to shortcutting. These align as expected. Note the non-zero origin y-axis.

the system call. Shortcutting attains around a 72% improvement in its latency.

LEBench `write()` test (Figure 4·22) The leftmost point on this graph corresponds to the average latency of 1-byte writes studied in the nanobenchmark section above. The rest of the line scans write sizes up to the 2-page boundary (8,192) bytes. We see Linux, kElevate (lowered), and kElevate (elevated) in close correspondence across write buffer sizes. We also notice a dip in latency at the integer page size buffer size (4,096 and 8,192 bytes, respectively).

The shortcutted cases have much lower latency, seeming to preserve a constant separation from the baselines across write sizes (consistent with the constant amount of work eliminated from the syscall path). At 1-byte, the Shortcut and In-Source Shortcut lines appear to be coincident. When looking at large `write()` buffer sizes, the In-Source shortcuts perform slightly better than when using the shortcutting script

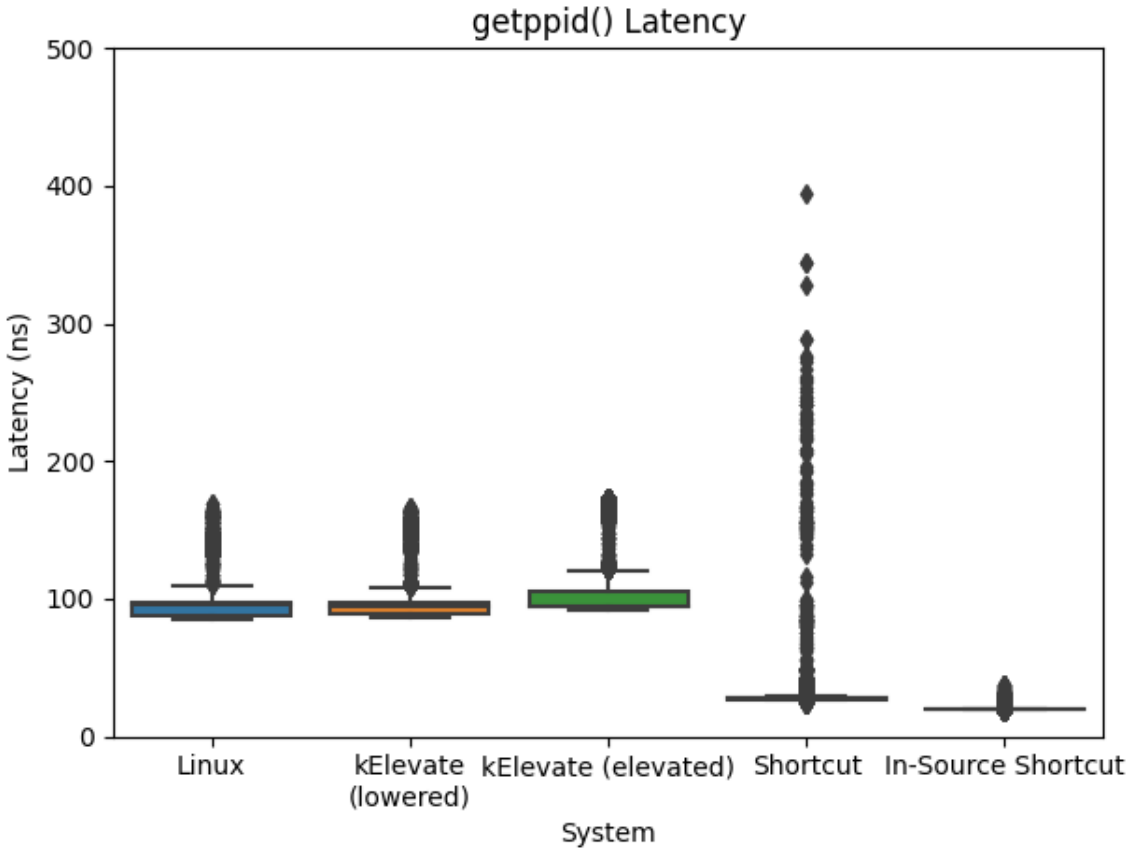

Figure 4·21: Latency boxplots for repeated calls to the `getpid()` system call.

through dynamic linking. A single-byte write sees 36% lower latency in the In-Source shortcut configuration than in baseline Linux. This percentage improvement drops as the write size increases because the constant amount of eliminated system call work takes up a proportionally shorter amount of the overall system call. This is likely because copying the write buffer from the user to the kernel context takes linear time.

LEBench `read()` test: (Figure 4·23) The `read()` component of this microbenchmark begins with the Linux baseline, kElevate (lowered), and the kElevate (elevated) cases tracking closely in terms of latency as a function of buffer size. These buffer sizes are both the size of the buffer provided to the `read()` system call and the number of bytes actually read into that buffer. We can see that both of the shortcutting cases,

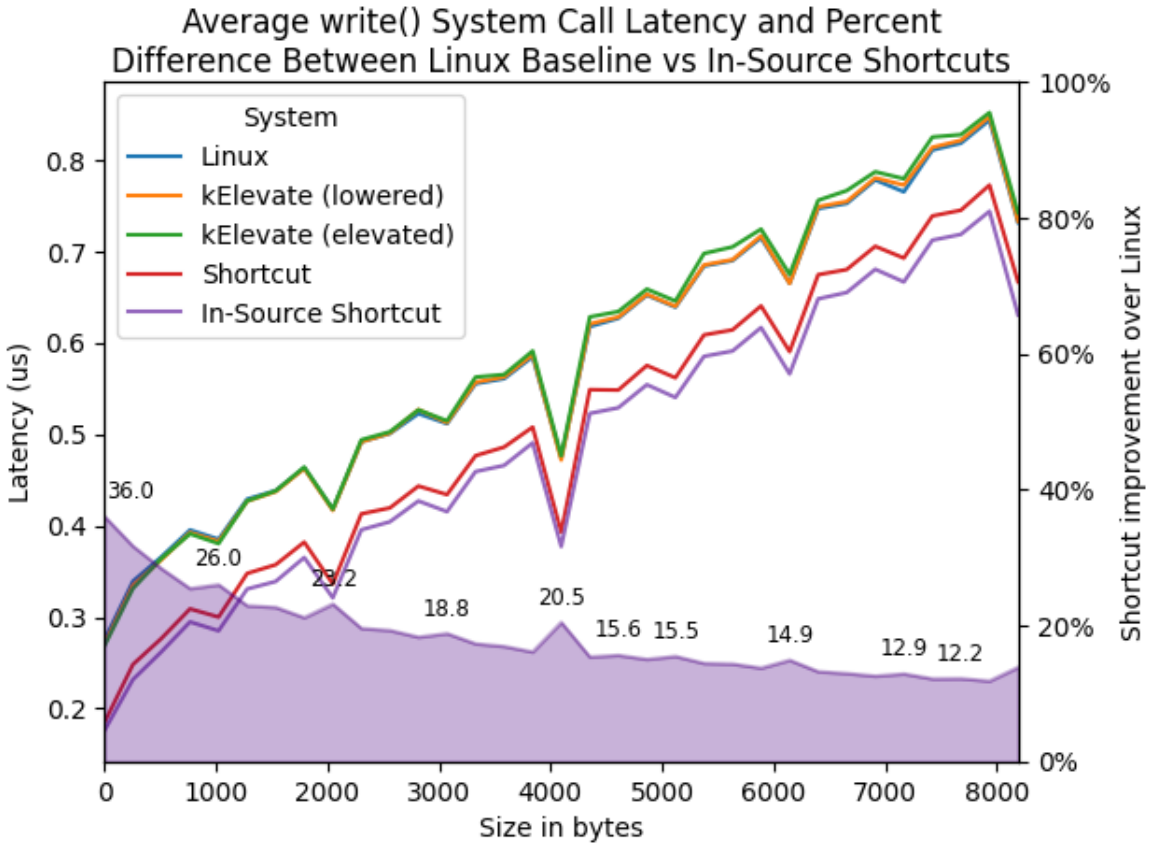

Figure 4·22: Comparison of average `write()` system call latency between system configurations across varying `write()` buffer sizes.

the dynamically linked shortcut case and the in-source shortcut cases, track closely with a significant latency improvement across the sizes measured. They range from the single-byte `read()`that's 37% faster than the Linux baseline to around 13% for an 8Kb `read()`. This performance improvement drops with the size of `read()` as the buffer grows in length. It is likely that this is because a constant amount of latency is removed from each system call, yet the amount of time to execute that system call grows as a function of the buffer size because (similar to the `write()`) the kernel is doing memory copies proportional to the size of that buffer, which incur a time complexity increase linearly proportional to the number of bytes `read()`.

LEBench `mmap()` test (Figure 4·24). The memory map, or `mmap()`, microbenchmark

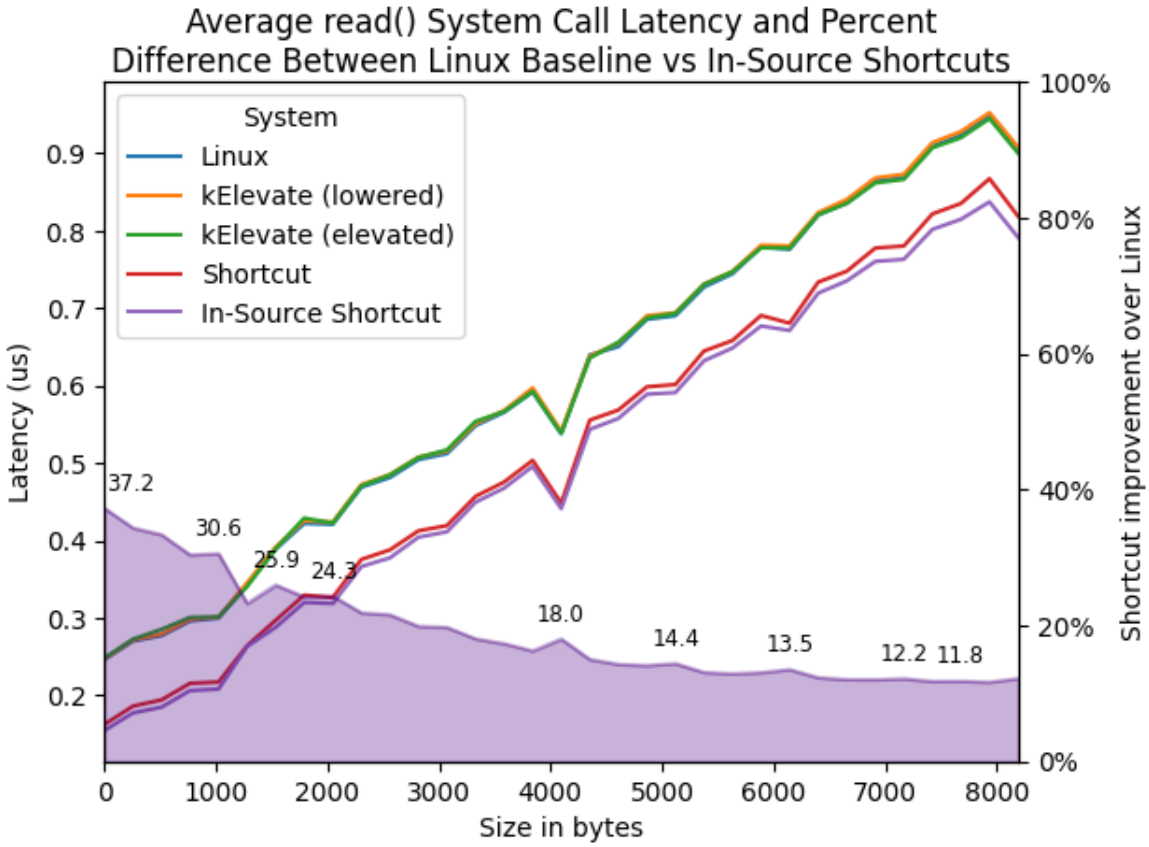

Figure 4·23: Comparison of average `read()` system call latency between system configurations across varying `read()` buffer sizes and bytes read.

shows four system configurations, the latencies achieved, and the percent improvement over the Linux baseline of the shortcutted case. The three configurations without shortcutting track fairly closely up to about a page size of 4,098 bytes. The shortcut case achieves a bit over a 20% improvement across the buffer sizes measured. Because the work done on an `mmap()` does not grow quickly with size, we do not see the same tailing-off effect of reduced improvement at larger allocation sizes, at least over the measured domain. We notice the kElevate (elevated) baseline taking slightly longer than the other baselines.

LEBench `munmap()` test (Figure 4·25). In this memory unmapping microbenchmark, we can see the Linux and kElevate (lower) configurations tracking closely across the

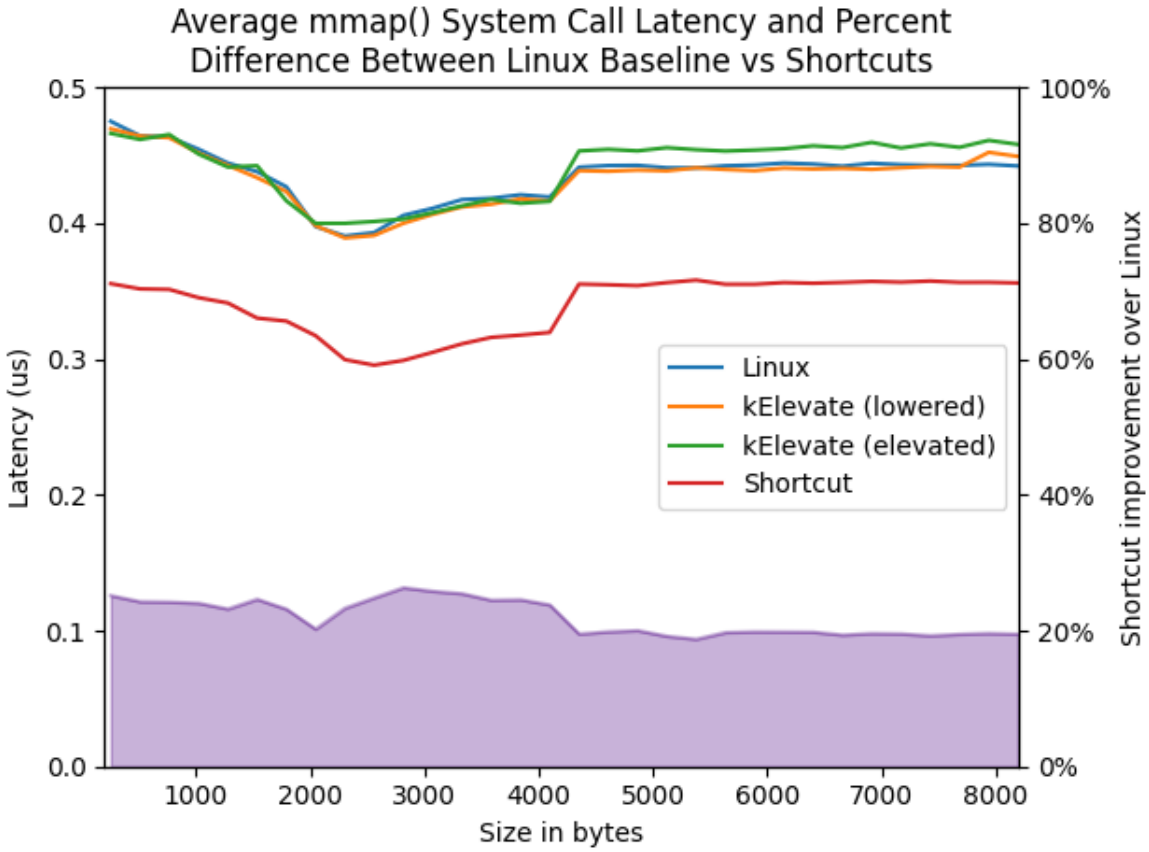

Figure 4·24: Comparison of average memory map, or `mmap()`, system call latency between system configurations across varying `mmap()` size.

different unmap sizes, with a discontinuity at the page size. The kElevate (elevated) configuration appears to run at slightly higher latency across the various memory unmap sizes.[16] The percentage improvement achieved with shortcutting is less than what is seen for the `mmap()` case. On average `munmap()` takes longer than `mmap()`; therefore, the percentage reduction in system call latency is lower, given the elimination of a constant amount of work from each call.

Microbenchmark Discussion: We conclude that shortcutting appears to be a viable approach to optimizing the performance and energy consumption of applications that follow the profile of these system call heavy workloads. It seems shortcutting

[16]Futher investigation is required to explain this effect if it persists.

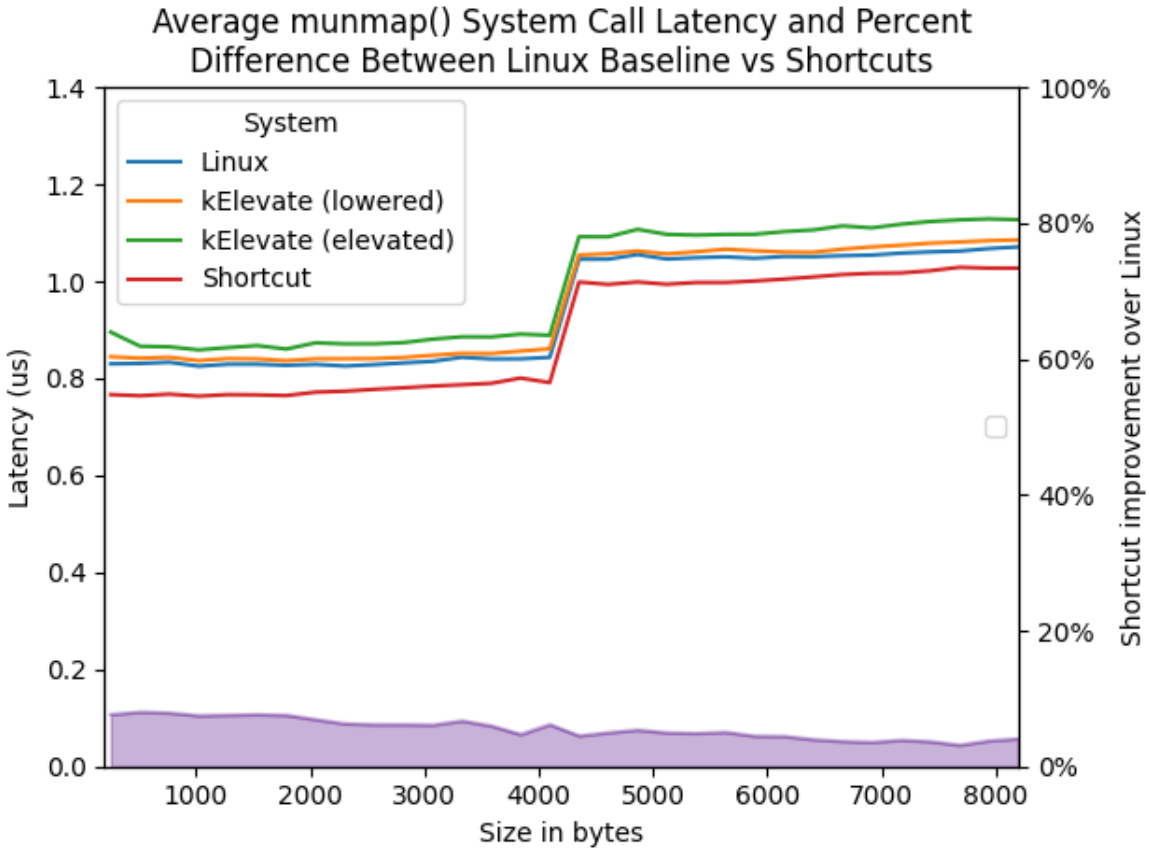

Figure 4·25: Comparison of average memory un-map, or `munmap()`, system call latency between system configurations across varying `munmap()` size.

removes a constant amount of latency from system calls. Proportionally, this affects short system calls more, e.g., 1-byte `write()`s see around 37% speedup, but even sizable 2 page (8,192-byte) `write()`s see a 13% speedup.

On different hardware, UKL achieved slightly better latency improvements baremetal, e.g., 47% average improvement on single-byte `read()`s, and 43% on single-byte `writes()`s. One difference is that UKL statically links the application with glibc and the kernel. This reduces dynamic linking overhead, whereas our kElevate approach uses indirect jumps (even for In-Source Shortcuts).

Returning to our questions: Did simply introducing support for kElevate change the performance profile of the kernel? The `write()`, read, `mmap()`, `getppid()`, and CPU

bound loops results suggest no change, however, but the `writeLoop` nanobenchmark shows a 2.7% difference. We do not believe the nanobenchmark difference will hold up, though it would take a more rigorous study to convince ourselves. We think the next-step would be to average across system reboots. We speculate that there is non-deterministic initialization in the kernel boot process, e.g., in the memory allocator, that would not show up when repeatedly running the application.

Does interposition slow down the application? The difference between the interposed configuration and the kElevate (lowered) configuration in Figure 4·16 suggest a slight slow down, Aa do the In-Source Shortcuts outperforming the Shortcut configurations.

Does running application code with privilege affect the performance characteristics of the application? The `read`, `write`, and CPU bound task graphs suggest not, but the `mmap()` (greater than 1-page size) and `munmap()` graphs indicate there may be some difference. A careful study of the `munmap()` loop, especially at larger sizes, may be interesting. Looking at kernel backtraces and taking into account statistics provided by hardware counters may shed some light.

Is ABI-compatible application shortcutting via dynamic linking a viable optimization strategy? The significant throughput, latency, and energy improvements observed in the Shortcut configuration compared to the Linux baseline demonstrate that ABI-compatible shortcutting via dynamic linking is indeed a viable optimization strategy. It's worth considering if these actually translate to real applications, which we will study next.

What's the most we can get from shortcutting when we allow application modification? The In-Source Shortcut configuration, which eliminates the overhead of dynamic linking, achieves the highest performance gains, suggesting that allowing application modification can further enhance the benefits of shortcutting.

In summary, the microbenchmark results provide strong evidence for the effectiveness

of shortcutting in improving throughput, latency, and energy consumption for system call-heavy workloads, without significantly altering the kernel's performance profile. The approach appears to be viable both through dynamic linking and with application modification.

Application-level Benchmark, Redis

Redis

To assess the impact of kElevate-enabled shortcutting on application performance, we evaluated Redis, a popular networked key-value store, on baremetal hardware, and in virtualization.

Baremetal Running baremetal on the "server" hardware (described in Section 4.2.3), we used two physical servers, sending network traffic from one to the other. The client was a Linux node, and the Redis server ran either Linux or the kElevate kernel. We compare three system configurations: 1) Linux 5.14.0 baseline, 2) Redis running on a kElevate kernel with shallow shortcuts, and 3) Redis on a kElevate kernel with deep shortcuts.

We started the Redis server on the Linux kernel with:

```
$ taskset -c 1 bash -c 'redis-server --protected-mode no --save ''
--appendonly no'
```

Then we sent "warmup" traffic to the node using the `memtier_benchmark`(mem,) tool to initialize the system:

```
$ taskset -c 1,2,3,4,5,6 memtier_benchmark -s 172.16.97.208 -p 6379 -P
redis -t 6 -c 25 -n 5000 -R 1000 --ratio=0:1 -d 1
```

Then we sent the real workload for which we measured throughput:

```
$ taskset -c 1,2,3,4,5,6 memtier_benchmark -s 172.16.97.208 -p 6379 -P
redis -t 6 -c 25 -n 40000 -R 1000 --ratio=0:1 -d 1 > output.txt
```

These client command lines configure the server to use six threads and 25 connections per thread. The requests are of random sizes, and 40k requests are sent per client. We found that pinning threads to sets of cores helped reduce aggregate throughput variance.

In the shortcutting cases, we used the shortcut script to affect either shallow shortcutting:

```
$ taskset -c 1 bash -c ‘shortcut.sh -be -s "write->__x64_sys_write" -s
"read->__x64_sys_read" --- redis-server --protected-mode no --save ‘’
--appendonly no’
```

or deep shortcutting:

```
$ taskset -c 1 bash -c ‘shortcut.sh -be -s "write->tcp_sendmsg:<write_addr>"
-s "read->tcp_recvmsg:<read_addr>" --- redis-server --protected-mode
no --save ‘’ --appendonly no’
```

System	Xput (GETS/s)	Improvement
Linux	29,3201	-
Shallow SC	32,5793	11.1%
Deep SC	35,5401	21.2%

Table 4.2: Comparison of Linux, shallow shortcut, and deep shortcut performance.

The results, summarized in Table 4.2, show that shallow shortcutting led to 11% higher throughput, and deep shortcutting nearly doubled that improvement to 21%. We emphasize that these performance gains are achieved without modifying or recompiling Redis, demonstrating the effectiveness of kElevate's ABI-compatible shortcutting approach. These are comparable to UKL's 12% and 26% improvements to Redis throughput for shallow and deep shortcutting respectively.

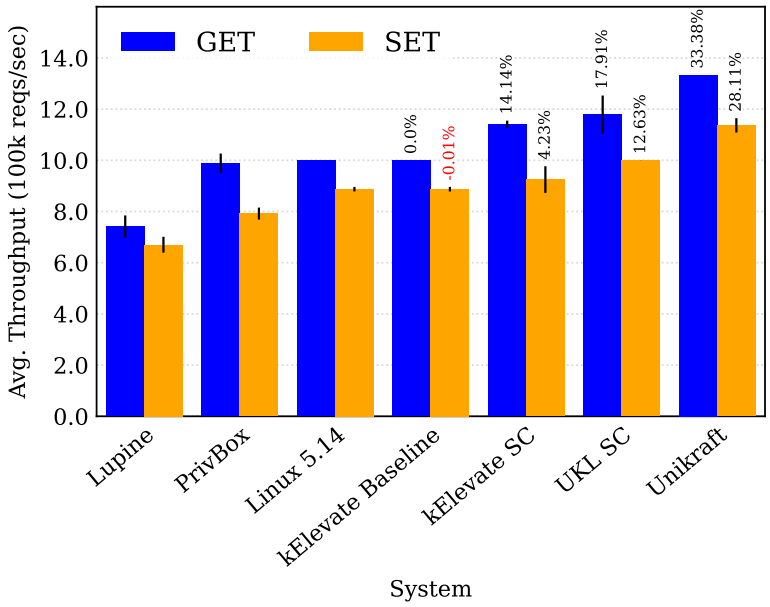

Figure 4·26: Throughput of various systems when running Redis as measured in transactions per second. Percentage improvement is shown for kEvevate, UKL, and Unikraft as performance improvement over Linux 5.14. Note that Unikraft crashed when using 100 concurrent connections, so it was run with 50.

Virtual Machines For our virtualized experiments, we measured the performance of a kElevate-enabled system against Lupine (Kuo et al., 2020b), PrivBox (Kuznetsov and Morrison, 2022b), UKL (Raza et al., 2023), and Unikraft (Kuenzer et al., 2021). We use the same version of Redis as PrivBox, so our Linux baselines, all the kElevate configurations, and PrivBox use this version. We did not update Redis for Lupine since it uses a custom build system, and Unikraft requires its own custom implementation of the Redis protocol. While PrivBox required Redis to be re-compiled, a common binary was used for Linux and kElevate, demonstrating ABI compatibility even when an application is shortcutted.

We adopted the evaluation methodology used by Unikraft, with redis-benchmark [17] on the host machine, pinned to cores sharing the socket with the guest with pipelining of 16, where the benchmark executed SET commands followed by GET and each system was booted fresh for every run. To maximize throughput on our more modern hardware, we increased the benchmark from 50 to 100 concurrent connections, and to two million transactions. For PrivBox and Lupine, we compared against the most performant version of that system. For PrivBox, we used the full system with no instrumentation; for Lupine, we used the DJW + KML patch kernel configuration with Redis as a trusted binary.

We used six different setups when testing a kElevate-enabled kernel: 1) a standard Linux userspace on top of a kElevate kernel (kElevate Baseline) to measure any impact of the mechanism on processes that do not make use of it, 2) pass-through mode which uses the shortcut script to launch Redis and preloads the shortcut library, but does not make use of interposition, elevations, or direct kernel interaction, 3) interposing mitigations enabled which is the same a passthrough with the mitigations requires using kElevate enabled, 4) elevated which adds executing the `kElevate()` system call

[17] Our redis-benchmark invocation was `taskset -c 3,4,5 redis-benchmark -{}-csv -q -{}-threads 3 -n 2000000 -c 100 -h ${IP} -p 6379 -d 3 -k 1 -t set,get -P 16`.

before starting Redis, 5) shallow shortcuts as described in Section 4.2.2, and finally 6) deep shortcuts as described in Section 4.2.2. We found that there was little to no difference between kElevate Baseline and the other kElevate systems. kElevate with shallow shortcuts did show showed around 5% (Get) performance improvement, while deep shortcuts (SC in Figure 4·26) 14% Get and 4% Set.

Figure 4·26 shows the throughput of the best-performing configuration for each system. We find that deep shortcuts outperformed Linux, Lupine, and PrivBox. kElevate has the advantage of being a very simple kernel addition, which makes rebasing the changes fairly simple, avoiding being abandoned like Kernel Mode Linux/Lupine and using standard tooling. Direct comparisons between Lupine and PrivBox with the other systems are difficult because they are based on different kernel versions and use different compilers. Our experience with Unikraft crashing highlights a difficulty when using from-scratch systems, often they are not as well tested as Linux or Redis.

2.4 Shortcutting Discussion & Conclusion

This section demonstrated that Dynamic Privilege enables application optimization through the technique of shortcutting system calls. By directly invoking kernel system call handlers and even deeper kernel paths, applications can significantly reduce the overhead of system calls on performance-critical paths. The kElevate mechanism allows shortcutting unmodified dynamically linked Linux binaries (preserves ABI-compatibility). Shortcutting can be applied to every instance of a particular system call in an application, or be targeted to one or more specific call sites.

Microbenchmarks show that enacting specialization via shortcutting achieves simultaneous throughput, latency, and energy optimization. They demonstrated that shortcutting removes a constant amount of overhead from system calls. While this reduces the average system call latency of shorter system calls by the largest percent, such as `getppid()` (72%) and 1-byte `write()`s (37%), longer system calls like 2-page

`writes()` still see a speedup (13%). We showed that shortcutting generalized across a set of system calls, including `getppid`, `read`, `write`, `mmap`, and `munmap`. Finally, we showed that deep shortcuts improved Redis GET throughput by 21% on a baremetal server, and 14% in virtualization.

While not applicable to every application, shortcutting demonstrates how providing applications with hardware privilege can enable profound system optimization. The nature of Dynamic Privilege makes it easy to integrate these optimizations with the existing application and kernel software and to explore them incrementally. By granting applications the ability to invoke kernel code paths, Dynamic Privilege allows developers to redefine the application-kernel interface at runtime using application-level tools.

Chapter 5

Adapting General-Purpose OSs to Support Elevated Threads

This chapter explores our approach to addressing practical challenges when utilizing Dynamic Privilege to elevate threads of standard user-level software running on Linux. Linux was not developed with the notion of elevated application threads in mind. There are critical differences between the application and kernel execution models that must be addressed when running non-trivial application software in privilege.

Our approach to addressing these differences in execution models involves creating low-level OS modifications called "adaptors" which modify the kernel to make it more compatible with elevated application-level software. Interestingly, adaptors can be implemented using Dynamic Privilege itself without the need for external mechanisms. We have shown that sophisticated software stacks like the multi-threaded key-value store Memcached can be executed in privilege with the addition of only two adaptors, both modifying Linux's interrupt handling for page faulting.

Our experience using Dynamic Privilege on Linux has led us to hypothesize that resolving the differences between the application and kernel execution models is a matter of configuring the kernel to make it "aware" of the existence of elevated application threads, rather than requiring a significant rewrite. While we do not claim that our provided adaptors are "complete" or sufficient to run any application with hardware privilege, our experience suggests that applying multiple independent adaptors to the OS at runtime is a viable approach for supporting a larger set of applications or even

running any Linux application with privilege. We believe a set of such general-purpose adaptors will be small (e.g., fewer than ten), and this iterative approach allows developers to select how much support they require at a fine granularity, avoiding the "nothing works until everything works" problem of kernel construction.[1]

This chapter comprises the following subsections (see Chapter 5). We discuss the fundamental differences between the kernel and application execution models and present two specific challenges Section 5.1. Next, we propose adaptors as an incremental approach to modifying Linux to support elevated application threads Section 5.2. We describe two specific adaptors that address the raised issues Section 5.3 and introduce interrupt interposition as a common implementation method Section 5.4. We take a deep dive into implementing a kernel error-checking adaptor in Section 5.5. Finally, we summarize the key points and conclude in Section 5.6.

[1]Some may prefer special purpose adaptors that optimize for e.g., performance. Ultimately, libraries of adaptors for particular use cases may proliferate.

Kernel vs Application Execution Models

Differences in execution models

Challenges running elevated apps in Linux

Adaptor Solution

Low-level OS modifications for execution model differences

Address challenges with Dynamic Privilege and kElevate

Two Adaptor Examples

Kernel Correctness Checks

Stack Starvation

Interrupt Interposition Technique

Common solution for both example challenges

Intercepts interrupts, redirects to custom handlers

Page Fault Error Checking Adaptor Details

Intercepts faults, modifies error code to deceive kernel

Uses kElevate to replace IDT, execute functions, allocate memory

Discussion and Conclusion

Potential to address other challenges like SMEP/SMAP

Automating deployment with adaptAll and idtTool

Figure 5·1: Section Organization.

1 Differences in Execution Models & Challenges Running Elevated Threads

As mentioned in Chapter 2, there are fundamental differences in the execution models of applications and the Linux kernel. In this section, we describe two particular challenges arising from these particular differences. In later subsections, we use adaptors to address these differences, enabling application threads to execute with privilege

on Linux. In our prior work on UKL(Raza et al., 2023), we enumerated a larger set of differences between the application and kernel execution models. Significant differences include: 1) the assumption that the kernel will run with hardware privilege, while the application will not; 2) the kernel will execute in the upper half of the virtual address space, and the application will run in the lower half; 3) the kernel will use fixed size pinned stacks, while applications use dynamically paged and potentially large stacks; 4) the kernel can access the application portion of the virtual address space, but the application cannot access the kernel portion.

1.1 Memory Management Challenges

Here, we address two concrete challenges in memory management that appear when executing application threads with privilege. The first challenge is kernel assertions or "correctness checks" that correctly identify kernel invariants that elevated threads break. One example is on the page fault path, where the kernel checks the privilege mode of an application thread when faulting. The second challenge is the kernel correctly diagnosing "stack starvation," which should never happen according to the Linux design. Our adaptors address these challenges by modifying the IDT to point to a new interrupt handler that accounts for elevated threads before forwarding to the original handler. This modification technique is described as "interposition" on the interrupt handlers.

Kernel Invariant Correctness Checks

A check in the page fault handling path ensures that a process will be killed if a page in the user portion of the address space is accessed while executing with hardware privilege. This check helps enforce system assumptions and prevents privileged code from accidentally executing user code or being tricked into executing arbitrary user code with elevated privileges. However, the Dynamic Privilege model purposefully

allows a user thread to run in the privileged state while executing user code, which contradicts the default assumptions. As a result, this page fault check severely restricts an elevated user thread, allowing it to execute only the code on user pages already established in the page table; accessing any other pages would lead to the process being killed. The following describes how this manifests in the x86 implementation of Linux.

The Linux kernel runs with hardware privilege in the upper half of the virtual address space, while applications run without privilege in the lower half. The kernel strictly enforces these invariants, occasionally with hard-coded assertions. The Linux kernel incorporates sanity checks on critical memory management paths as a defensive programming technique. One such check is found in the page fault handler, which uses the fault address (delivered by hardware upon a page fault exception in Control Register 2) to differentiate between user and kernel faults.

Once a user page fault is identified, a sanity check ensures that the error code–data pushed onto the stack by the hardware-—correctly encodes entry from user mode. This check safeguards against Return Oriented Programming (ROP) style attacks, where a user tricks the kernel into executing its application code while still in privilege. Failure to pass this check triggers an "Oops" in the kernel and the termination of the offending process. Checks of this form introduce a challenge for elevated threads executing in supervisor mode, which should be able to handle page faults.

1.2 Stack Starvation

In Linux, page faults to the stack are assumed never to occur when executing kernel code and, thus, when executing with privilege. This invariant is ensured by executing kernel code using a "kernel stack." Kernel stacks are of fixed length, and their pages are pre-established and pinned in the page tables; thus, access to them can never induce a page fault.

On x86, a page fault induces the pushing of an exception frame onto the current stack. If the page of the stack that the exception frame is being written to is not in the page table, an x86 double fault is triggered. The Linux double fault handler has been written to crash the system if this scenario occurs, as by design, it should not. This is true for two reasons. First, as stated above, the pages of kernel stacks will always be present in the page tables. Second, Linux programs the x86 hardware such that switching from user privilege to kernel privilege will include a stack switch to a kernel stack. As such, if a page fault occurs while attempting to access a user stack page, the fault itself will be handled on a kernel stack, and a double fault will be avoided.

More generally, these scenarios fall under the broader OS design and implementation issues regarding recursive faulting and "stack starvation" This issue is further discussed in KML, UKL, and Privbox (Maeda and Yonezawa, 2003; Raza et al., 2023; Kuznetsov and Morrison, 2022a).

Again, Dynamic Privilege challenges this design and the coded assumptions. Once elevated, a thread, now running with privilege and using a standard pageable user stack memory region, may very well trigger a page fault on stack access but not induce a privilege switch, thus leading to a double fault and crashing the system. This explicitly violates the assumption that executing with privilege is equivalent to using a non-pageable region of memory for the stack.

2 Adapting the Linux Kernel to Support Elevated Threads

Our approach to modifying a general-purpose OS like Linux to support the execution of application threads with hardware privilege involves creating a set of low-level OS modifications called adaptors. We apply adaptors at runtime, using them to incrementally resolve the differences between the application and kernel execution

models that may otherwise crash the system, or prevent elevated application threads from making progress. This incremental approach avoids the "nothing works until everything works" problem, enabling developers to focus on their workloads. This problem arose because of Dynamic Privilege, and we will solve it using Dynamic Privilege.

Adaptors provide software engineering flexibility by offering a range of options for addressing compatibility issues between elevated application threads and the Linux kernel. Multiple adaptors could address the same underlying incompatibility, achieving different trade-offs e.g., in terms of performance, security, and engineering effort. There need not be a single "ultimate" set of adaptors; instead, the set of adaptors can evolve as new use cases emerge and new kernel features are developed. Suppose the Linux kernel community decides to support Dynamic Privilege as a first-class feature; in that case, they may incorporate some well-tested and broadly applicable adaptors directly into the kernel source tree. In Section 5.3, we overview adaptors that address the two named challenges in Section 5.1, including two different adaptors for the correctness checking issue and discuss their trade-offs.

The adaptor approach embodies the policy component of the policy–mechanism separation we described when exploring the kElevate mechanism in Chapter 3, Section 3.1.1. kElevate provides the fundamental mechanism for an application thread to attain hardware privilege, while adaptors encode the policy choices governing how privileged threads should resolve compatibility issues with Linux. This separation allows the kernel to be modified to support elevated threads without adding complexity to the core kElevate mechanism. Our prototype exemplifies this by encapsulating the existing set of adaptors as user binaries and attendant shell scripts. As such, no additional kernel modifications are needed, and using Dynamic Privileges is as easy as running the scripts. The next paragraph describes these scripts.

We run a single shell script to prepare a Linux system for executing elevated threads, `adaptAll.sh`. `adaptAll.sh` applies all of our adaptors to each CPU core. This process allows elevated and non-elevated application threads to coexist on the same system. We believe that continuing to expand the set of available adaptors will allow an increasingly large set of applications to run with privilege on Linux.

3 Adaptor Examples

In the Section 5.1, we discussed two specific challenges that arise when executing elevated application threads on Linux: kernel correctness checks and stack starvation. In the following subsections, we will overview adaptors that target these issues, demonstrating how adaptors can resolve the differences between the two execution models incrementally. This subsection is only an overview of the adaptors we use. Section 5.4 introduces interrupt interposition, and Section 5.5 provides a deeper dive into our usage of interrupt interposition to enable the kernel correctness check adaptor.

3.1 Kernel Correctness Checks

We've established that kernel code paths include assertions that verify certain system invariants are upheld at runtime. One is a check that a page fault originating in the user portion of the address space indeed occurred when the processor was in user mode. This will catch page faults of elevated threads, generating a kernel "Oops," which will terminate the process. Here, we overview two adaptors that address the issue. The first uses binary rewriting to write `nop` instructions over the offending code path. The second uses interrupt interposition to fixup the error code before the page fault handler even runs. While either is sufficient to address the issue, they come with different trade-offs.

Nop Slide Adaptor Overview

The first adaptor we created for the kernel correctness check issue uses a straightforward hack known as the "NOP slide." This technique involves overwriting the offending check in the kernel's page fault handler with no-operation (NOP) instructions, eliminating the check. To accomplish this, we developed a Bash script called `noper` that uses `readelf` to determine the address range of the unwanted check using DWARF debugging information from the kernel binary. It then invokes a C program named `writeBytes`, which uses kElevate to gain the necessary privileges to overwrite the check with NOP instructions directly in the kernel's text section. Since the targeted kernel memory pages may not have write permissions, `writeBytes` also traverses the page table and modifies the permissions as needed, restoring them after the modification is complete.[2] While this approach effectively eliminates the kernel correctness check, it removes it for all application page faults in the system, not just those from elevated threads, and requires debug information for the kernel.

Interrupt Interposition Adaptor Overview

A more nuanced adaptor addresses this issue using interrupt interposition to intercept page faults before they reach the kernel's handler. The adaptor first creates a new Interrupt Descriptor Table (IDT) and modifies the entry corresponding to page faults to point to a custom handler. This custom handler checks if the current process is elevated and that the fault error code matches expected conditions (e.g., a fault in the user half of the address space). If these conditions are met, the handler modifies the error code by setting the user bit, making it appear as if the fault originated from user mode. The adaptor then invokes the original page fault handler to ensure proper fault

[2]Notice this is a simple but reasonably general set of foundational tools. We have used them to do more interesting binary rewrites, e.g., modifying assembly code to support new inputs to kernel functions.

servicing. This deception allows the kernel's page fault handler to proceed unmodified without terminating the process. This allows for special handling only for elevated threads, without changing handling for lowered threads. We will study this adaptor in Section 5.5 after building some background on interrupt interposition in Section 5.4.

3.2 Stack Starvation

The kernel doesn't fault on stack pages; applications do. The kernel only configures a stack change to occur when page faulting from user mode. This leads to stack starvation and unrecoverable double faults (see Section 5.1). To mitigate this issue, the stack starvation adaptor allows the initial stack page fault to result in a double fault, but intercepts the double fault handler to recover safely. The adaptor uses interrupt interposition to replace the double fault handler in the IDT with a custom handler. This custom handler invokes the page fault handler. This only works because Linux configures the double fault handler to run on a known safe Interrupt Stack Table (IST) stack designed to handle critical errors. By servicing the initial stack page fault on the IST stack, the adaptor prevents the recursive fault scenario and allows the elevated application thread to continue executing. We developed this technique working on UKL(Raza et al., 2023). This adaptor demonstrates that it is possible to address a critical incompatibility with a tiny modification that reuses existing kernel code.

4 Interrupt Interposition & the IDT

This subsection first provides background information on the Interrupt Descriptor Table (IDT), then describes how we modify it, using interrupt interposition to implement adaptors. The tools in this section will demonstrate the following Dynamic Privilege Capabilities:

1. Reading and writing privileged system registers,
2. Reading and writing kernel memory, and
3. Finding kernel symbol addresses
4. Execute kernel code (e.g. allocating kernel memory)
5. Installing new kernel code
6. Modifying existing kernel code

4.1 Background: The IDT

Interrupts are event-driven mechanisms initiated by hardware and software to signal the processor that immediate handling is required. Linux's interrupt subsystem centers on the IDT, a data structure used by various architectures, such as x86_64 and ARM64, to determine the appropriate handler for each interrupt or exception (see Figure 5·2). On x86_64 systems, each CPU core has its own Interrupt Descriptor Table Register (IDTR) pointing to the core's IDT, and Linux sets all IDTRs to point to the same IDT in memory. In general, when Linux is ported to non-x86 architectures, it tends to implement the x86 design as closely as possible. Given this we focus our IDT discussion on Linux's x86 implmentaiton.

The IDT contains descriptors for each interrupt.[3] Descriptors specify the control flow destination for the corresponding interrupt service routine. These descriptors also include information such as which x86 segment selector in the Global Descriptor Table should be used, the x86 Interrupt Stack Table (IST) stack to be used (if any), the x86 gate type (indicating whether re-execution of the interrupted instruction should be

[3]These events are often colloquially referred to as "interrupts" (because they interrupt software code paths). However, there are times when it's useful to be more precise about their taxonomy. More formally, an "interrupt" is an asynchronous event delivered by off-CPU hardware (like a timer or I/O device), while an "exception" is an event generated by the CPU itself (like a trap, fault, or abort).

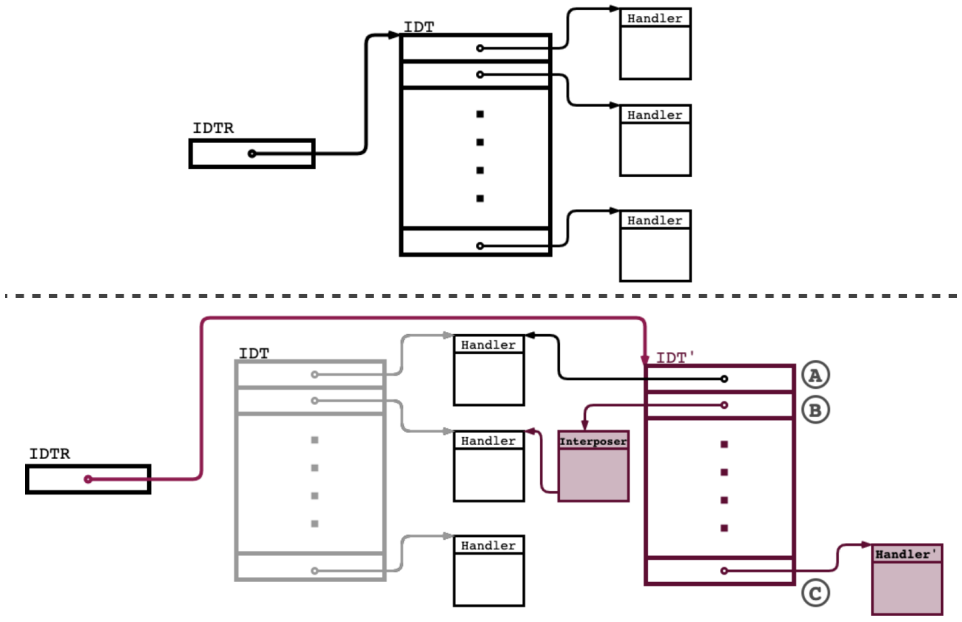

Figure 5·2: x86 IDT and our IDT adaptor design. Top, the IDTR points to a page in memory containing the IDT. The IDT includes descriptors holding configuration state and pointers to interrupt handlers. Bottom, our approach to interrupt interposition involves copying the old IDT to IDT', then updating select descriptors to branch to some custom interposition code before continuing to their original handlers (as seen in B).

attempted), and the descriptor privilege level (the minimum privilege level required to invoke the exception). Although rarely used, IST stacks provide known safe stacks for printing bail-out messages when encountering an unrecoverable error or if the kernel stack is corrupted or exhausted. In addition to the IDT, other system states, such as x86 Control Registers (e.g., CR4) and page tables (pointed to by CR3), are closely involved in interrupt handling, particularly those related to memory management.

4.2 Interrupt Interposition:

Interposition is a general technique that intercepts control flow and redirects it, typically to custom handlers, before invoking the intended software path. The interrupt interposition technique is one way we have explored implementing adaptors that affect interrupt handling (like page faulting). By interposing on specific interrupts, such as

page faults, adaptors can modify the system's behavior to support the execution of elevated application threads on Linux.

Our approach to achieving interrupt interposition on the IDT involves copying the original IDT and updating the descriptors of the targeted interrupts to point to custom interposition handlers. These custom handlers can perform additional checks, modify incoming data, or execute specific code before invoking the original interrupt handlers. The final step is to install the modified IDT. On x86, this involves updating a control register called the IDTR on each core to point to the new IDT. This effectively redirects the targeted interrupts to the custom handlers, allowing the adaptors to intercept and modify the interrupt handling process. Interrupt interposition, as described, can be achieved with a set of elevated user primitives. In particular:

Perhaps unsurprisingly, we have found that interrupt interposition is a powerful tool for building adaptors. Furthermore, the decomposition of the implementation into usable primitives, which can be constructed using Dynamic Privilege without dependency on interrupt interposition, illustrates the viability of using Dynamic Privilege incrementally. In the following subsection, we explore one adaptor that utilizes interrupt interposition in detail.

5 Page Fault Error Checking Deep Dive

In this subsection, we dive into our adaptor's implementation details that address the kernel correctness check issue on the page fault path (Overviewed in Section 5.3). By leveraging interrupt interposition on the IDT and the capabilities of kElevate, the adaptor intercepts page faults, modifies the error code, and deceives the kernel into believing that the fault originated from user mode. This allows the kernel's page fault handler to proceed unmodified, enabling proper page fault functionality.

5.1 Modifying the Error Code

The following discussion is x86-specific. When a page fault occurs, the hardware places the offending address into control register 2 (CR2) and pushes an error code onto the stack. The kernel expects a specific bit in the error code to indicate user mode for addresses in the lower half of the address space. However, in elevated mode, this bit indicates supervisor mode. Our interposition (see Section 5.5.3 and Section 5.5.2 for details) checks if the thread was elevated and if the fault type was as expected. If both conditions are met, the interposer flips the bit to indicate user mode, deceiving the kernel. This allows the unmodified page fault code to run as expected. Other architectures may use hardware to enforce this invariant. Alternative approaches will need to be explored.

To complete the interposition, we must call the original page fault handler. We use kElevate to execute the Linux kernel function `kallsyms_lookup_name`, passing in a symbol name as a string and receiving the symbol's address as a return value.[4] We use this function to determine the location of the original page fault handler and then call it from our interposer.

5.2 Allocating the IDT in the Kernel

We allocate memory for the new IDT and interposition handler in the kernel, giving our IDT the same lifetime as the kernel and preventing memory revocation, which would have occurred when the process dies. After studying the various kernel page allocator functions, we found that `vzalloc` was suitable for our needs because it allocates virtually contiguous zero-filled memory. Using the `kallsyms_lookup_name` function to find `vzalloc`, we allocated a page of memory in the kernel and used the `glibc` function `memcpy` to copy the kernel IDT to the new page.

[4] One might simply read the handler address from the IDT, but this allows us to rerun the tool.

We wrote the handler code and packaged it into a user program (the idtTool described next). The handler code is designed to be copied onto kernel memory similarly allocated using `vzalloc`. This assembly code fixes up the error code (described in Section 5.3.1). Finally, we loaded the address of the copied interposer code into the newly allocated IDT and installed it by updating the IDTR. A difference with this approach is instead of having one IDT, we have one IDT per core. Changing to multiple IDTs is not required; we can revisit this policy decision and even switch to using a single core (making sure to address the issues of atomicity). We have implemented the process for allocating and manipulating the IDT into a program we call the idtTool, which exploits Dynamic Privilege, and is described in detail in the next section.

5.3 The idtTool and Automation

The idtTool is a critical component of our approach to implementing the necessary adaptors that allow application threads to execute on Linux with privilege. At present, the idtTool is x86 specific. It is an application-level program written in C, and as the name suggests, it is a tool for manipulating the IDT, changing the way it responds to interrupts and exceptions. The key uses of kElevate involve executing privileged instructions, executing kernel functions to allocate kernel memory, and copying kernel memory onto those allocated pages. The main instructions for modifying the IDT Register (IDTR), which is a per-core pointer to an IDT, are the `lidt` and `sidt` instructions.

We show the help menu in Figure 5·3. An example shows how repeated invocations of the tool accumulate to copy, modify, and install a new IDT at runtime: 1) getting the kernel address of the IDT; 2) copying the IDT onto a kernel page; 3) copying an interposition onto a kernel page; 4) modifying a vector of the IDT at a given address; 5) installing an IDT that lives at a particular kernel address by swinging the IDTR.

```
[kele@rfc1918 bin]$ ./idt_tool -h
./idt_tool:
options:
        -a <addr>: address of idt, current loaded assumed if not provided
        c:         copy idt return ptr to copy on kern pg
        g:         get current idtr
        h:         print this help msg
        i:         install idt (swing idtr)
        m <ist_enable|ist_disable|addr:0xaddr>: modify idt entry
        p:         print
        v <dec#>:  vector number for print / modify
        z <df|tf>: which mitigation to copy to kern page

examples:
        taskset -c 1 ./idt_tool -g
        taskset -c 1 ./idt_tool -c
        ./idt_tool -z tf
        ./idt_tool -a ffffc90000986000 -m addr:0xffffc9000098d000 -v 14
        taskset -c 1 ./idt_tool -a ffffc90000986000 -i

df mitigation workflow:
        taskset -c 0 ./idt_tool -g
        taskset -c 0 ./idt_tool -c
        ./idt_tool -z df
        ./idt_tool -a ffffc900002ef000 -m addr:0xffffc90000317000 -v 8
        taskset -c 0 ./idt_tool -a ffffc900002ef000 -i
[kele@rfc1918 bin]$
```

Figure 5·3: Help screen for the idtTool, used for modifying the IDT. Serves as a low-level interface for implementing adaptors, among other features.

Use of the idtTool is wrapped by the interposing_mitigator.sh shell script (see Figure 5·4). This script supports modifying the IDT on a given core in 4 ways. Two of these implement the stack starvation and correctness check adaptors. The other two are for unrelated functionality that has to do with interposing on debug exceptions and the int3 exception. Figure 5·5 shows a verbose printout of running the interposing_mitigator to inoculate core 0 against the stack starvation problem. The interposing_mitigator applies our two adaptors to a given core. Externally, the adaptAll.sh script drives the interposing_mitigator, applying it to all of the cores on the system in parallel.[5]

1. Obtain the existing IDT from the memory location `0xffffc90000d2d000`.

2. Allocate a kernel page and copy the original IDT onto it at the memory location `0xffffc90000ded000`.

[5]While we have used the idtTool daily on many baremetal environments, including 80-core servers, further testing for possible race conditions and other issues would be welcome.

3. Allocate another page and write the new double fault handler onto it at the memory location `0xffffc90000df5000`.

4. Modify the copy of the IDT such that vector 8, the double fault handler, points to the page where the new handler is located.

5. Finally, "install" the new IDT by pointing to core 0's IDTR at the new page.

```
[kele@rfc1918 recipes]$ interposing_mitigator.sh -h
script for performing interrupt interposition mitigations
flags: -m "tf\df" -t <core> -d <debug>

examples:
./interposing_mitigator.sh -m tf -t 1 -d
./interposing_mitigator.sh -m df -t 1 -d
./interposing_mitigator.sh -m i3 -t 1 -d
./interposing_mitigator.sh -m db -t 1 -d
[kele@rfc1918 recipes]$
```

Figure 5·4: The interposing mitigator, applies two of our adaptors, one for the kernel check on the page fault path, and the other to prevent the stack starvation issue.

These tools exhibit a number of properties that Dynamic Privilege makes possible: 1) accessing kernel registers; 2) Allocating memory with kernel persistence properties (that an application will write to); 3) Incorporating new code into kernel code paths; 4) modifying incoming data to circumvent kernel logic.

6 Conclusion

In this chapter, we introduced adaptors as a practical approach for incrementally modifying the Linux kernel to support executing application threads with hardware privilege. Adaptors address key differences between the kernel and application execution models that can cause issues when running elevated application threads on Linux.

```
[kele@rfc1918 recipes]$ ./interposing_mitigator.sh -m df -t 0 -d
Apply df mitigation on core 0

about to get initial idt:
taskset -c 0 /home/kele/Symbi-OS/Tools/bin/recipes/../idt_tool -g
initial idt: ffffc90000d2d000

allocate kern page
copy page ffffc90000d2d000
taskset -c 0 /home/kele/Symbi-OS/Tools/bin/recipes/../idt_tool -c
onto page ffffc90000ded000

allocate kern page
copy df handler
/home/kele/Symbi-OS/Tools/bin/recipes/../idt_tool -z df
onto page 0xffffc90000df5000
with scratchpad page 0xffffc90000dfd000

point new idt ffffc90000ded000 vector 8 at 0xffffc90000df5000:
/home/kele/Symbi-OS/Tools/bin/recipes/../idt_tool -a ffffc90000ded000 -m addr:0xffffc90000df5000 -v 8
ffffc90000ded000
hdl installed

make new idt live
Load IDTR with ffffc90000ded000:
taskset -c 0 /home/kele/Symbi-OS/Tools/bin/recipes/../idt_tool -a ffffc90000ded000 -i
new idt should be live
old is ffffc90000d2d000
```

Figure 5·5: The verbose printing of running the interposing_mitigator script, which drives the idtTool program to prevent elevated stack faults from triggering double faults–which would lead to the kernel panicking.

We described two specific challenges: kernel correctness checks on the page fault path and stack starvation due to recursive page faults. To resolve these issues, we developed adaptors that leverage Dynamic Privilege mechanism via interrupt interposition on the IDT. We dove into the implementation details of the page fault error-checking adaptor, showcasing the use of kElevate and the idtTool for manipulating the IDT. We do not argue that all adaptors must be implemented using interrupt interposition, but we do believe this approach to applying adaptors to other general-purpose OSs and other architectures. This technique relies on common low-level aspects that most operating systems handle in similar ways. For example, interrupts are the core mechanism used by operating systems to define the execution environment. The major general-purpose OSs of our day are monoliths that run applications and kernels in opposite ends of the same address space. Similarly, page allocators are a standard mechanism found in most operating systems, making the techniques portable across

different platforms.

This experience left us with the following insight: adapting a general-purpose OS to support elevated threads is less about extensive rewrites and more about targeted adjustments that make the kernel "aware" of the elevated application threads. This approach opens up new possibilities for running a wide range of applications with hardware privilege on Linux. Adaptors can be used in a one-off approach to support a single application, or a set of adaptors could together significantly expand the scope of applications capable of benefiting from elevated execution.

Chapter 6

Conclusion

The conventional wisdom in system design dictates a static divide between code that runs with hardware privilege and code that runs without. This separation is ingrained deeply in operating systems and application software design and implementation. We proposed a new OS model, Dynamic Privilege, to challenge this long-held assumption, enabling threads to make independent transfers between privilege levels during their lifetimes.

This research recognizes the fundamental differences between kernel and user models of execution; each has its unique strengths. Kernels are powerful, with unrestricted access to hardware resources, while user applications benefit from a rich software ecosystem. This work demonstrates that with the addition of a simple mechanism, kElevate, we can implement Dynamic Privilege on a general-purpose OS and unite these two worlds, enabling a more fluid model of software interaction. Through case studies, we have explored allowing user threads to elevate their privilege and access kernel resources on demand. Further, Dynamic Privilege provides a novel approach to specialization and prototyping by opening up the kernel as a toolbox for user applications to exploit

The rest of this chapter proceeds as follows: Section 6.1 reviews core research contributions, Section 6.2 discusses possible future work where Dynamic Privilege may connect with other parts of the field, and Section 6.3 contains some personal remarks on the work.

1 Summary of Key Findings and Contributions

We break our summary of the key findings and contributions of this dissertation into the following parts: defining the Dynamic Privilege model, developing the kElevate mechanism, the adaptor methodology, exploring Dynamic Privilege via case studies, and frameworks and conceptual contributions.

1.1 Defining The Dynamic Privilege Model

The Dynamic Privilege OS model (see Section 3.1) introduces a new perspective on accessing hardware privilege. The critical insight is that it decouples a change in hardware mode of execution from the change in control flow, as prescribed by all existing transfers: system calls, interrupts, and their associated returns. With this capability, we can explore the ramifications of lowering kernel threads and elevating user threads.

Dynamic Privilege offers a fine-grained level of control over privilege because it can be accessed on a per-thread basis and for any temporal scale (down to executing a single instruction). Defining this model close to the hardware simplifies implementation and lays the foundation for building arbitrary higher-level policies that exploit it. The model itself does not suggest any particular implementation. See Section 6.2.1 for a discussion of how this model may generalize to the hypervisor layer and beyond.

1.2 Development of the kElevate Mechanism

To demonstrate the feasibility and practical implications of the Dynamic Privilege model, we developed the kElevate mechanism (see Section 3.2). kElevate is a new system call we added to the Linux OS, allowing application threads to toggle their execution mode. By invoking kElevate, an application thread can enter into privilege, access the extended ISA, and interface with kernel code paths and data structures

without context-switching. We designed this interface primarily to allow application threads to toggle privilege. While we have prototyped a mechanism to lower kernel threads, we define this outside the scope of this dissertation, addressing them in future research Section 6.2.

The kElevate Mechanism is implemented in under 200 LoC. It consists of the following:

1. A 2-bit addition to the `task_struct`, Linux's process/thread control block to track the privilege status of a thread, and if the thread is migrating cores
2. A conditional check on the system call return path to return with/without Privilege
3. The kElevate system call itself
4. A credential check to verify permission to execute the syscall
5. An architecture-specific check on the context-switch path to update the application's segment register on core if the thread is migrating

As discussed in Section 6.1.2, adding a mechanism to an existing operating system to support the dynamic elevation of an application thread's privilege is relatively straightforward. However, this modification can lead to potential issues due to the operating system's initial static privilege model. When an application thread executes with hardware privilege, it can violate the system's fundamental assumptions. Moreover, defining what such a thread can "safely" do can be challenging. For instance, as explored in Section 5.3, even the thread's ability to execute instructions within its own address space can lead to challenges.

Dynamic Privilege can be implemented by clean-slate OSs or retrofitted into existing OSs, as we did by implementing kElevate in Linux. The kElevate mechanism is

simple because it is defined close to the hardware implementation and designed to be lightweight and devoid of policy decisions, deferring these to tools that utilize the mechanism. kElevate essentially serves as a software emulation of a future hardware mechanism. For more on these points, see Section 6.2.2.

1.3 Adaptor Methodology

To address the challenges of running privileged application threads on a general-purpose operating system like Linux, we introduced the concept of adaptors (see Chapter 5). Adaptors are low-level modifications to the operating system that bridge the gap between the application and kernel execution models. The adaptors we implemented allow privileged threads to execute for a set of applications, overcoming the incompatibilities that would otherwise prevent these threads from running.

The adaptor methodology is an incremental approach to modifying the OS to support Dynamic Privilege. Rather than requiring extensive changes to the kernel, adaptors target specific incompatibilities and provide targeted solutions. This approach enables a gradual adoption of Dynamic Privilege, avoiding the "nothing works until everything works" problem, allowing developers to focus on their specific use cases and requirements.

As discussed in Section 6.1.2, kElevate is a low-level mechanism that intentionally defers policy decisions to higher-level tools, allowing for flexibility in its use. Adaptors are an example of such higher-level tools, encoding specific policies to address the challenges of running privileged threads on Linux. In Section 5.3, we described two different adaptors that tackle a specific issue, demonstrating that developers can choose the adaptors that best suit their needs. As the Dynamic Privilege ecosystem matures, we anticipate the development of general-purpose adaptor sets that can support a wide range of elevated processes (see Section 6.1.3).

1.4 Exploring Dynamic Privilege via Case Studies

This dissertation considered multiple case studies, each utilizing privileged threads from the application context. These case studies consist of user-level applications compiled with standard toolchains and interpreters but can execute privileged instructions, read kernel memory, and exercise kernel code paths. They demonstrated multiple properties, including:

1. Low-level access to privileged hardware resources
2. The ability to access kernel data structures and code paths
3. High-level kernel access including use of macros, types, and dynamic linking
4. Enacting performance and energy optimization via shortcutting system call paths
5. Optimizing unmodified dynamically linked applications (preserving ABI compatibility)

The primary property these case studies demonstrate is unification across the application-kernel boundary, where elevated threads can access privilege and, thus, kernel resources. We demonstrated that this access works not just in principle but offers similar convenience that kernel programmers or module developers would expect. This includes access to kernel abstractions like macros, types, data structures, and subroutines. We drive this point home by performance and energy optimizing the Redis networked key-value store with zero porting effort. The compiled dynamically linked binaries were entirely unmodified (ABI compatible), yet we could use standard dynamic linking interposition to shortcut general-purpose system call paths.

These case studies are motivated by our desire to address a fundamental asymmetry we observe between the rich application ecosystem and restricted kernel environment.

Humanity has developed a staggering amount and diversity of application-level software. A single version control hosting service, GitHub, reports hosting over 420 million repositories. Much of this application-level software consists of tools such as libraries, frameworks, languages, compilers, orchestration software, profilers, verifiers, analyzers, and debuggers. These tools reduce programmer effort and enable applications to meet multiply-constrained optimization targets.

Despite the diversity of OS kernel designs explored in the literature (addressed in Section 2.2.3), the major OSs have primarily settled on a single OS organization: the general-purpose monolithic OS. This seems peculiar, and we conjecture that it results from path dependency[1] and is a testament to how difficult it is to build system software. It is too expensive to explore far from this design point.

> **To be clear, we have no interest in adding complexity to OSs.** On the contrary, we believe Dynamic Privilege can significantly simplify OSs by, for example, moving policy out of OSs and into privileged servers (see Section 6.2.2). We hope for a world where we can leverage significant portions of the already-built application-level software infrastructure to explore system design. We hope this brings flexibility to existing systems and lowers the cost of exploring new OS designs and organizations.

1.5 Frameworks and Conceptual Contributions

In addition to the practical contributions of the Dynamic Privilege model and the kElevate mechanism, this work also makes significant conceptual contributions. We introduced the source-to-runtime (see Section 3.1.3) spectrum as a framework for analyzing the separation between application and kernel code throughout the software lifecycle. This framework provides a comprehensive view of the various stages, from

[1] By this, we mean inertia and resistance to change.

source code to executable formats and runtime execution, highlighting the opportunities for optimization and integration.

Further, we proposed a three-part framework for understanding the impact of Dynamic Privilege on OS design (see Section 3.3). This framework contrasts Dynamic Privilege with existing extension mechanisms and Unikernel architectures, arguing that only Dynamic Privilege delivers all three of the following properties: 1) compatibility with the application execution model, 2) kernel-level access, and 3) retention of procedural control. By positioning Dynamic Privilege within the broader context of OS design and extensibility, this framework provides a foundation for future research and development in this area.

Finally, in Section 2.2 we documented the evolution of Dynamic Privilege as an outgrowth of our prior work, and other academic and industrial influences.

2 Future Work

We envision a future where Dynamic Privilege forms the foundation for various areas of research and industrially deployed systems. This section will explore potential connection points organized into the following main categories: extensions to the Dynamic Privilege model and alternative implementation Section 6.2.1; kernel architecture and design Section 6.2.2; application optimization Section 6.2.3; and security and reliability Section 6.2.4.

2.1 The Dynamic Privilege Model and Implementation

Here, we offer some possible extensions to the Dynamic Privilege model and its implementation.

Expanding and Fine-Graining the Dynamic Privilege Model: To scope the contribution, we studied Dynamic Privilege from the perspective of supervisor and user privilege levels, primarily utilizing privilege from the application context. Further ex-

ploration of lowering kernel code should be conducted (as mentioned in Section 6.2.4). However, the space extends much further than this. Some processors offer intermediate privilege levels, hypervisor levels, secure enclaves, and system management modes. These can all be explored for possible synergy. We've come this far; why not allow applications to modify the firmware directly?

Alternative Dynamic Privilege implementation: We are exploring an attempt to implement the mechanism solely through the kernel module mechanism, e.g., by using `ioctl` along with module patching of the critical kernel code paths. We hope to use this exploration to refine the mechanism-policy separation, implementing as much as possible outside the kernel.

Hardware support: We want to see Dynamic Privilege's elevate and lower functionality built into computer hardware. Changing privilege as a single instruction may be 100 to 1000 times faster than emulating it through a system call. Faster transitions allow for much less performance perturbation, enabling its practical use in many new contexts. Enabling this may include adding one or more bits to the hardware representation of the per-task state to track the current privilege status. Emulating this in a virtual machine monitor like QEMU for a RISC-V processor may be a viable starting point.

2.2 Kernel Architecture and Development

Unifying Kernel Distribution: In practice, OS distributors often maintain multiple kernel versions and configurations for customers. This multiplies the testing matrix, leading to significant engineering and QA expenses. Dynamic Privilege may offer a unification to this process if it can be used to move kernel modification from build time to runtime, potentially taking significant kernel build time off the critical path. Microkernels have been studied extensively; they involve producing user-level servers that factor out much of the functionality typically found in monolithic kernels (see

Section 2.2.4). One of the significant costs associated with microkernels is the context-switching. Dynamic Privilege presents an opportunity to create "privileged servers" that need not incur the context-switch costs to the kernel.

These privileged servers provide a means to decompose a monolithic kernel into separable components, which are beneficial for testing, failure isolation, incremental updates, and runtime patching. Decoupled from kernel code, privileged servers can be written in high-level languages (see Section 6.2.4 for thoughts on securing these servers).

The Anykernel(Kantee et al., 2012) model encouraged building device drivers that could run anywhere: in the kernel, as user-level servers, and as library code for application usage. The goal was to ease the device driver development effort because it is easier to develop at the application-level than at the kernel-level. Dynamic privilege offers similar prototyping benefits, except it does not need to emulate privileged access.

We imagine "donor kernels" which allow the use of kernel code from other kernels, potentially embedded, e.g., an alternative network stack into a privileged application. Taking this to the extreme, a privileged monitor could unplug system resources like cores and channels on network interface controllers (NICs) and dedicate them to application-specific Unikernels such as UKL(Raza et al., 2023).

For a concrete example of the path toward utilizing donor kernels, consider the kernel bypass framework DPDK, the Data Plane Development Kit. DPDK apps run as user processes on top of Linux. A subset of cores are hot-unplugged from the kernel and assigned to this app. Lightweight user-space network stacks interact with user-level device drivers assigned by the kernel to NICs. DPDK apps demand resource dedication and expensive application porting. Resources are removed from kernel management and run in busy polling loops. The DPDK network stack is a de facto

donated kernel component; it is not part of Linux, yet it benefits an application running on Linux. We suggest using Dynamic Privilege to explore these system organizations and interpolate between them, with the added possibility of accessing privileged resources. See the following section, Section 6.2.3 for more on optimization.

ForkU: We are presently exploring using Dynamic Privilege to build a privileged monitor capable of transparently forking arbitrary user-level processes. We were able to implement this tool quickly because it reuses much kernel functionality. Unlike a traditional `fork()`, which immediately schedules the child to run, we are able to hold a cache of these children, waiting to run on demand. This "snapshot-replay" model is the backbone of computational caching strategies like the one we deployed in SEUSS(Cadden et al., 2020). Using hierarchical copy-on-write semantics through the fork tree enables memory sharing within these lineages. It can also be used for replay debugging.

Adaptor Coverage: The next natural step concerning adaptors is to extend coverage. We know some programs such as Firefox and GCC fail when we attempt to run them with privilege. Building out a set of adaptors that allow any Linux program to run elevated is a sensible and immediate approach to enabling wider use of the model. We suspect this process would make explicit more app-kernel differences that are presently unstated. Different OSs will require different adaptors, and multiple implementations of adaptors may be used on a single OS for specialization.

Kernel Interpreter: When a Python interpreter modifies the IDT, the Python interpreter "kernel" becomes the OS kernel in a sense. How far are we from writing kernel components as platform-independent bytecode? We believe a useful project to pursue would be the construction of a series of Jupiter notebooks that progressively build up the hardware data structures that allow it to act as an OS kernel, for some subset of the hardware.

Aligning OS and Application Worlds: Dynamic Privilege lifts the barrier between the application and OS. Our use of adaptors resolves some of the differences between these worlds. How many of these differences are simply different conventional choices that arose because of the independence of these codebases? Will the desire for tighter integration pressure interoperability? Will kernels be adjusted to remove these differences?

Exploring Dynamic Privilege on Other OSs: The other major general-purpose OSs seem to offer a very similar monolithic structure to Linux, with applications and kernel running on opposite ends of the same address space. Implementing Dynamic Privilege on them may be a straightforward task. What might be a greater challenge is exploiting the model on a microkernel that runs applications and the kernel in orthogonal address spaces. Separate address spaces may require rethinking the implementation for kernel state to remain accessible efficiently.

Clean-slate OS Implementation: Instead of retrofitting Dynamic Privilege into an existing OS, what would it look like to build an OS with this functionality from the start? Would it entail a race to get a trivial user space up, from which point user-level tooling is used to replace the kernel bootstrap?

2.3 Application Optimization

Kernel developers often treat applications as black boxes, software which they will not inspect or understand the inner workings of. Similarly, application developers do the same with kernels. The problem with this black-box abstraction is that it precludes co-optimization. Dynamic Privilege removes the barrier between the application and the kernel, enabling whole-system optimization. This newly unified software space offers opportunities to apply known optimizations and discover new ones.

Replace Kernel Watermarks with Externally Tunable Parameters: General-purpose kernels serve a wide range of applications on diverse hardware, from smartwatches

to supercomputers. Monolithic kernels like Linux embed significant policy decisions within the kernel, often using simple watermarks and limits to enforce these policies. For example, Linux's `__do_softirq()` function, which controls softirq processing (essentially interrupt bottom halves), will process softirqs until 10 have been processed or 2ms have passed. Code comments state that these limits were established via experimentation to balance latency and fairness. However, determining these values experimentally is difficult, as fairness and latency are not the only concerns. Factors like throughput, energy, SMP scalability, microarchitectural cache pollution, and debugging also matter. Further, the optimal values depend on various factors such as architecture, devices, user programs, libraries, client demand, and whether the kernel is running baremetal or in virtualization.

Instead of burdening kernel development with these policy decisions, we suggest addressing them at runtime. While limited support exists for profile-guided and runtime optimization via user-level tuning processes like Tuned(Hat, 2023), even in dynamic mode, these still use predefined rules and thresholds. We are presently exploring using Dynamic Privilege to enable reinforcement learning to adjust processor frequencies and NIC packet coalescing to affect whole system energy consumption and application-level performance targets(Dong et al., 2021). Generally, we recommend designs where a monitor runs separately from the kernel and can access and modify both the application and kernel parameters. It will likely operate best when presented with multiple tunable parameters that affect core functionality.

High-Performance Networking: In data centers, high-throughput and low-latency networking is crucial for applications such as software-defined networking, 5G, edge systems, and high-frequency trading. To achieve extreme optimization, custom OSs and bypass frameworks like DPDK (see Section 6.2.2) are employed. However, DPDK's high-frequency polling consumes significant energy.

Dynamic Privilege allows applications, such as HPC DPDK apps, to optimize energy consumption by directly controlling power states through issuing (privileged) WRMSR instructions that configure the system. This enables bringing research results(Dong et al., 2021) into application software, exploiting findings such as using polling with low P states and low C states. By empowering applications to manage power states at a fine granularity, Dynamic Privilege allows for balancing performance and energy efficiency based on current demands.

Real Time: Discussions with researchers in the field of Real-Time Operating Systems (RTOS) highlight a potential opportunity to exploit Dynamic Privilege. With Dynamic Privilege, it may be possible to reduce the variance in execution latency for paths that cross between application and kernel software. This approach could serve as a less complex and less invasive alternative to comprehensive solutions like the Linux Real-Time patches. Our results indicate the ability to reduce variance in latencies for executing system calls, including reducing maximum observed latencies. Although the spectrum of real-time requirements is long, Linux equipped with Dynamic Privilege becomes an attractive platform to develop for RTOS workloads.

Upward shortcutting: The case study on shortcutting (see Section 4.2) focused on "downward" shortcutting from the application into the OS. A symmetric "upward" shortcut opportunity exists going up from, e.g., network interrupts directly into executing application code. Our prior work on EbbRT demonstrated the value of stitching application logic onto the interrupt handling paths. Dynamic Privilege upward shortcuts can provide a fine-grain and selective approach to achieving similar optimizations in the context of Linux.

Direct access to privileged hardware: enables application-specific optimizations. With Dynamic Privilege, applications can control virtual memory translation, enabling custom memory management techniques like tailored garbage collection and page size

management. This allows for fine-grained optimizations based on the application's memory usage patterns. Additionally, applications can directly manage non-volatile memory (NVRAM), exploiting its persistence properties, as well as its status in the caching heirarchy, typically between DRAM and storage.

2.4 Kernel Security & Reliability

Dynamic Privilege may be used to improve kernel security and reliability. Recall that access to privilege via Dynamic Privilege uses the same credential check used by kernel modules, which are unlimited in their ability to change the kernel. While improvements to the Dynamic Privilege mechanism, such as incorporating cryptographic signing of binaries, may be beneficial, we believe the most interesting future work lies in evidencing or proving security properties of the policy contained in binaries that access privilege.

In this section, we challenge the notion that a kernel is only "secure" when unmodified. We will discuss an example in practical systems, SELinux, where modules and user programs enhance kernel security. The rest of the section considers future work suggesting techniques for improving the security of elevated processes and the kernel itself. These techniques include software verification, the sandboxing of kernel code, kernel patching, and integrating capability systems with kernel privilege.

Security is a free parameter: We have repeatedly encountered the erroneous assumption that Dynamic Privilege makes a kernel "insecure." The sentiment seems to stem from the idea that any tool that allows for kernel extension or modification only increases risk by increasing the attack surface of the trusted compute base. The existence of runtime kernel modifications that improve security easily disproves this assumption. Security is a free parameter for powerful mechanisms that allow for kernel modification, like modules and Dynamic Privilege. Security may increase, decrease, or stay the same depending on the policy a given extension implements.

Security-Enhanced Linux (SELinux) is a concrete example of a module-based runtime modification to the kernel that improves its security profile. SELinux is a kernel module and set of user space tools initially developed by the US National Security Agency (NSA). It was released in 2000 and upstreamed into the Linux kernel in 2003. SELinux uses Mandatory Access Controls to restrict the ways that "actors" (like users and processes) can affect "objects" (like files) according to some policy. SELinux is one example of a class of Linux Security Modules (LSMs) that improve over the bare kernel's security profile. Readers may also consider other LSMs like AppArmor and TOMOYO, as well as kernel mechanisms like containers, namespacing, seccomp, and cgroups. Note how, in these examples, the module mechanism itself—which enables unfettered access to the kernel—is not the "weak" link. Rather, it is simply the way to boot-strap and initialize a secure computing environment from a general-purpose software base.

Community review: While the academic conversation about security centers on threat models, proofs, and software and hardware-based safety mechanisms, these are merely tools, not complete solutions in the real world (see Section 1.6 for an expanded discussion). The integrity of large software projects, such as the Linux kernel, is maintained through significant human effort. This is achieved through traditional code review and testing by maintainers and the community at large. While there are many interesting approaches to securing the use of this powerful mechanism, we believe human oversight will be the main one, at least for as long as humans are the ones writing code.

We envision a future where communities develop around the use of Dynamic Privilege. Just as the Linux kernel community maintains thousands of modules, we anticipate communities supporting numerous binaries and frameworks that utilize Dynamic Privilege. Existing software projects may provide libraries and open-source soft-

ware or distribute closed binaries to protect intellectual property, similar to Nvidia's closed-source drivers. As with all software projects, these communities may provide compatibility between their projects or fork to build custom frameworks.

Integrating capabilities: Modern commodity computer architectures utilize translation hardware (page tables and TLBs) for both virtual-to-physical address mapping and coarse-grained access protection. In contrast, capability-based computer systems employ hardware/software tooling to provide fine-grained access control. We propose using capabilities to separate code and data into multi-level privilege classes, allowing kernel and application software to be granted explicit permissions to access code and data at various levels of this hierarchy. This approach offers the potential for more granular control over system resources and privileged operations, perhaps integrating with verification and other tools.

Sandboxing: Kernel code has software vulnerabilities. Binaries utilizing Dynamic Privilege will too. One approach to enhancing security (as well as improving prototyping and debugging) is to minimize the amount of code running with privilege. This can be achieved by running all code without privilege in a monitor, only elevating privileges when privileged instruction execution or memory access triggers a fault. These faults provide a hook-point where the monitor has the opportunity to determine precisely what resources and functionality the privileged code can access. Dynamic Privilege can ease the development of such a monitor.

Verification techniques: Dynamic Privilege applications can incorporate verification techniques employed by other systems, such as static and dynamic instrumentation mechanisms. eBPF, KML, and Privbox utilize various tools to restrict privileged applications. eBPF runs a verifier before producing kernel modules, while KML uses Typed Assembly Language for binary-level type checking, and Privbox employs compile-time instrumentation. These systems balance functionality, programmer ef-

fort, and performance overheads, making high-level languages and software tooling essential for effective verification. However, these verification strategies have significant limitations, including the 3 listed prevent privileged instruction execution. There is much work to do in this space, but we believe it should be done at the application level.

Live Kernel Patching: Kernel patching is a viable approach to addressing kernel security issues without rebooting the system(Poimboeuf, 2014). These approaches typically utilize kernel module-based modifications but can involve direct binary modification. Our adaptor solution to the text fault issue utilized kernel patching (see Section 5.3.1). Dynamic Privilege offers the opportunity to use high-level languages and tooling to construct and apply kernel patches.

3 Concluding Remarks

Dynamic Privilege is a project that grew over multiple years and through the contributions of many individuals. As the project's creator, I'll share a few concluding remarks on how I think about the work at the end of the day.

The term "Dynamic Privilege" can be broken down into its constituent parts. "Dynamic" comes from the Greek root "dynamikos," meaning powerful. The physics community adopted "dynamic" to refer to time-varying systems, contrasting static systems. The word "privilege" has its roots in the Latin "privus," meaning private, and "lex/leg," meaning law. It refers to a law that applies to an individual or a group, not the population at large. I think of elevated application threads in this sense—they've taken on power for now, specifically a power that not every thread will normally obtain.

Symbiosis: In the biological sense, symbiosis is a relationship formed between organisms of different species as they interact. "Symbiosis" is a compound word borrowed

from Greek, meaning "living together." I think about Dynamic Privilege as removing a wall separating application and kernel ecosystems. These ecosystems evolved largely independently, so we have not seen the co-optimization that will arise when they can live together. While we didn't use this term in the dissertation, it is peppered through our codebases.

There's a Systems Party, and Everyone's Invited: When people ask me what I do at parties, I tell them I build operating systems, and they excuse themselves to find someone interesting to talk to. When programmers ask me what I do, I tell them I build operating systems, and they either say, "Wow, you must be a masochist," or, "Oh, I hated that class." These responses bum me out because I feel they are missing out on a beautiful world, and because I feel misunderstood.

We refer to ourselves as "plumbers," with the word doing double duty: highlighting the essential nature of the work but suggesting this low-level work is dirty and something others would not want to do. Some system developers take a misguided sense of pride in the fact that so many others find their area unapproachable and uncomfortable to engage in. They may dismiss high-level tools of the application world as crutches that "real programmers" don't need. But perhaps we can use Dynamic Privilege to unify our software, inviting everyone to help fill in the missing intermediate space between our two worlds (see Figure 6·1). To misquote Richard Feynann, "There's plenty of room in the middle."

Figure 6·1: Kernel development can feel Sisyphean. Applications quickly take flight but lack the power to affect the system. Maybe when they meet, we'll build better systems and have more fun doing it. Image credit: DALLE-2 and Romey Sklar.

I hope this research is a step toward a world where application-level programmers are more comfortable contributing to system solutions, and grizzled system programmers are happy to reach for application-level tools when they improve productivity. I hope students will use Dynamic Privilege to explore low-level systems and feel empowered to contribute.[2] When I tell a new programmer I build operating systems, I hope they'll say, "I loved that class."

[2] And have shorter PhDs.

References

RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. `https://github.com/RedisLabs/memtier_benchmark`. (Accessed on 05/07/2021).

(2021). Solo5 - a sandboxed execution environment for unikernels. `https://github.com/solo5/solo5`. Accessed on 2021-10-7.

Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. (1986). Mach: A new kernel foundation for unix development. In *USENIX Summer*.

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (1991). Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 95–109, New York, NY, USA. Association for Computing Machinery.

Arch Linux (2020). iouring(7) – asynchronous i/o facility. Accessed: 2024-04-12.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA. ACM.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: Safe user-level access to privileged {CPU} features. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 335–348.

Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. (2014). {IX}: A protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 49–65.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. (1995). Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA. Association for Computing Machinery.

Bryant, R. and O'Hallaron, D. (2016). *Computer Systems: A Programmer's Perspective*. Always Learning. Pearson.

Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. (2020). Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15.

Campbell, R. and Tan, S.-M. (1995). /spl mu/choices: an object-oriented multimedia operating system. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 90–94.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. (1995). Hive: Fault containment for shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 29(5):12–25.

Chen, J. B. and Bershad, B. N. (1993). The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 120–133, New York, NY, USA. Association for Computing Machinery.

Contributors, A. (2016a). Apache openwhisk. `https://openwhisk.apache.org/`. Accessed: November 13, 2023.

Contributors, F. (2020). libfuse. `https://github.com/libfuse/libfuse`. (Accessed on 09/07/2022).

Contributors, G. (2023). Options that control optimization. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. Accessed: November 15, 2023.

Contributors, L. (2006). Kernel probes (kprobes). `https://docs.kernel.org/trace/kprobes.html`. (Accessed on 09/07/2022).

Contributors, L. (2008). ftrace - function tracer. `https://www.kernel.org/doc/html/v4.18/trace/ftrace.html`. (Accessed on 09/07/2022).

Contributors, L. (2016b). ebpf - extended berkeley packet filter. `https://prototype-kernel.readthedocs.io/en/latest/bpf/`. (Accessed on 09/07/2022).

Contributors, L. (2019). KVM. `https://www.linux-kvm.org/page/Main_Page`. (Accessed on 01/15/2019).

Dahlin, M. D., Wang, R. Y., Anderson, T. E., and Patterson, D. A. (1994). Cooperative caching: Using remote client memory to improve file system performance. Technical Report UCB/CSD-94-844, EECS Department, University of California, Berkeley.

Dong, H., Arora, S., Awad, Y., Unger, T., Krieger, O., and Appavoo, J. (2021). Slowing down for performance and energy: An os-centric study in network driven workloads.

Eldridge, S., Waterland, A., Seltzer, M., Appavoo, J., and Joshi, A. (2015a). Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 99–112.

Eldridge, S., Waterland, A., Seltzer, M., Appavoo, J., and Joshi, A. (2015b). Towards general-purpose neural network computing. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, page 99–112, USA. IEEE Computer Society.

Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. (1995a). Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA. ACM.

Engler, D. R., Kaashoek, M. F., and O'Toole Jr, J. (1995b). Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266.

Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. (1995). Implementing global memory management in a workstation cluster. *SIGOPS Oper. Syst. Rev.*, 29(5):201–212.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. (1999). Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings Operating System Design and Implementation (**OSDI'99**)*, pages 87–100, Berkeley, CA, USA. Usenix Association.

Github ("2023"). torvalds/linux. `https://github.com/torvalds/linux`.

Gray, J. (1988). *The Transaction Concept: Virtues and Limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Gregg, B. (2016). The flame graph. *Commun. ACM*, 59(6):48–57.

Hat, R. (2023). Chapter 3. Tuned.

Heiser, G. and Elphinstone, K. (2016). L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1).

Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P., and Kozyrakis, C. (2021). Ghost: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA. Association for Computing Machinery.

Itoh, J.-i., Yokote, Y., and Tokoro, M. (1995). Scone: Using concurrent objects for low-level operating system programming. *SIGPLAN Not.*, 30(10):385–398.

Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. *SIGOPS Oper. Syst. Rev.*, 31(5):52–65.

Kantee, A. (2012). Flexible operating system internals: The design and implementation of the anykernel and rump kernels. In *The Design and Implementation of the Anykernel and Rump Kernels*.

Kantee, A. et al. (2012). Flexible operating system internals: the design and implementation of the anykernel and rump kernels.

Kernel-Contributers (2022). Kernel modules. `https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html`. (Accessed on 09/07/2022).

Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. (2006). K42: Building a Complete Operating System. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA. ACM.

Kuenzer, S., Bădoiu, V.-A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., et al. (2021). Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394.

Kuo, H.-C., Williams, D., Koller, R., and Mohan, S. (2020a). A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

Kuo, H.-C., Williams, D., Koller, R., and Mohan, S. (2020b). A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15.

Kuznetsov, D. and Morrison, A. (2022a). Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA. USENIX Association.

Kuznetsov, D. and Morrison, A. (2022b). Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Liedtke, J. (1996). Toward real microkernels. *Commun. ACM*, 39(9):70–77.

Linux man-pages project (2021). `https://man7.org/linux/man-pages/man7/vdso.7.html`. (Accessed 28 Oct, 2022).

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. (2013). Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472.

Maeda, T. and Yonezawa, A. (2003). Kernel mode linux: Toward an operating system protected by a type theory. In *Annual Asian Computing Science Conference*, pages 3–17. Springer.

Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X., and Vahdat, A. (2019). Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA. Association for Computing Machinery.

Mathieu Acher (2019). Learning the Linux Kernel Configuration Space: Results and Challenges. `https://inria.hal.science/hal-02342130/document#:~:text=Owing%20to%20the%20huge%20complexity,configuration%20space%20of%20the%20kernel`. (Accessed on 11/9/2023).

McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA. USENIX Association.

Mosberer, D. and Peterson, L. L. (1996). Making paths explicit in the scout operating system. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA. USENIX Association.

Patterson, D. A. and Hennessy, J. L. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Perez, D., Cubuk, E. D., Waterland, A., Kaxiras, E., and Voter, A. F. (2016). Long-time dynamics through parallel trajectory splicing. *Journal of Chemical Theory and Computation*, 12(1):18–28. PMID: 26605853.

Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

Poimboeuf, J. (2014). Introducing kpatch: Dynamic Kernel Patching.

QEMU (2019). `https://www.qemu.org/`. [Online; accessed 9-January-2019].

Raza, A., Unger, T., Boyd, M., Munson, E. B., Sohal, P., Drepper, U., Jones, R., de Oliveira, D. B., Woodman, L., Mancuso, R., Appavoo, J., and Krieger, O. (2023). Unikernel linux (ukl). In *Proceedings of the Eighteenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, United States.

Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. (2019). An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA. Association for Computing Machinery.

Rojas, R. (1996). *Neural Networks - A Systematic Introduction*. Springer-Verlag, Berlin.

Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.

Schatzberg, D., Cadden, J., Dong, H., Krieger, O., and Appavoo, J. (2016). Ebbrt: A framework for building per-application library operating systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 671–688.

Seltzer, M. I., Endo, Y., Small, C. A., and Smith, K. A. (1997). Issues in extensible operating systems. *Harvard Computer Science Group Technical Report TR-18-97*.

Shen, Z., Sun, Z., Sela, G.-E., Bagdasaryan, E., Delimitrou, C., Van Renesse, R., and Weatherspoon, H. (2019). X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA. Association for Computing Machinery.

Small, C. A. and Seltzer, M. I. (1994). Vino: An integrated platform for operating system and database research. *Harvard Computer Science Group Technical Report*.

Soares, L. and Stumm, M. (2010). FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC. USENIX Association.

SRL, D. S. (2005). Kallsyms(8). `https://nixdoc.net/man-pages/Linux/man8/kallsyms.8.html`.

Waterland, A., Angelino, E., Adams, R. P., Appavoo, J., and Seltzer, M. (2014). Asc: Automatically scalable computation. *SIGARCH Comput. Archit. News*, 42(1):575–590.

Williams, D., Koller, R., Lucina, M., and Prakash, N. (2018). Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211, New York, NY, USA. Association for Computing Machinery.

Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. (2014). The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News*, 42(3):457–468.

Thomas Unger

tunger10@gmail.com | linkedin.com/in/tunger10 | github.com/tommy-u

Dr. Unger designs and implements user-friendly systems software that improves performance and energy efficiency. An experimental scientist at heart, he relies on rigorous measurement to guide cross-layer optimization. His doctoral work introduced "Dynamic Privilege," a novel OS model. This model enables developers to create highly optimized execution paths by selectively collapsing the traditional boundary between application and kernel code. He has strong interest and experience in optimizing application and system performance using machine learning.

Education

Boston University: Ph.D. Boston, MA
Computer Science: Systems *Sept. 2015 – May 2024*

Boston University: B.A. Boston, MA
Computer Science and Physics *Sept. 2010 – Dec. 2014*

Experience

Ph.D. Research, Selected Projects Sept. 2015 – Present
Boston University *Boston, MA*

- **Dynamic Privilege:** Designed and implemented a new kernel mechanism that allows Linux application threads to take on the full power of the operating system at runtime. Led a team of 5 graduate students to demonstrate that Dynamic Privilege allows specialization, resulting in simultaneous optimization of throughput, latency, and power (20-25% for each). This optimization was accomplished with zero application modification (ABI compatibility) to real networked workloads in virtualization and on bare-metal servers.
- **Unikernel Linux (UKL)**: Co-led design and implementation of a Linux configuration option allowing for a single, optimized process to link with the kernel directly and run at supervisor privilege. Implemented "deep shortcuts," the most aggressive optimization the project achieved: 26% throughput improvement for Redis. Presented the peer-reviewed work at a conference and invited talk venues. UKL remains an active Red Hat project.
- **SEUSS**: Used "snapshotting" to accelerate serverless workloads, resulting in an order of magnitude drop in function cold-start times, 15x better memory density, and a better security profile. Peer-reviewed paper with 135+ citations.
- **XFILES**: Built tools to debug Neural Network backpropagation on FPGA accelerators. Workshop paper.
- **ASC**: Autoparallelized single-threaded C code via Neural Network-based speculative execution. Extended Neural-Network library to allow for "expanding networks." Built constant-sized network inputs using bloom filters.

Collaboratory Partner Engineer Jan. 2024 – Present
Red Hat *Boston, MA*

- Conducted interviews with Red Hat's senior technical leadership to identify opportunities to connect research to the real-world problems of developing and testing an OS distribution.
- Continued Mentoring Red Hat software engineer interns.

Software Engineer Intern Sep. 2018 – Dec. 2023
Red Hat *Boston, MA*

- Carried out Ph.D research with significant input from Red Hat mentors.
- Collaborated with and mentored many Red Hat interns and students.

Software Engineer Intern Summer 2015
MorphoTrust USA *Billerica, MA*
- Implemented a "fast path" passport classifier prototype using machine learning.

Undergraduate Research 2012-2014
Boston University *Boston, MA*
- Tested & repaired "waveform digitizers," high-frequency DSP hardware for a particle physics experiment at the Paul Scherrer Institute.

Publications

Published Peer-Reviewed Papers

- A. Raza, T. Unger, M. Boyd, E. Munson, P. Sohal, U. Drepper, R. Jones, D. Bristot de Oliveira, L. Woodman, R. Mancuso, J. Appavoo, O. Krieger, "Integrating Unikernel Optimizations in a General Purpose OS" arXiv:2206.00789v1 [cs.OS] 1 Jun 2022
- J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In Proceedings of EuroSys '20: The European Conference on Computer Systems (EuroSys '20). ACM, New York, NY, USA

Published Workshop Papers

- S. Eldridge., T. Unger, M. Sahaya Louis, A. Waterland, M. Seltzer, J. Appavoo, and A. Joshi, "Neural Networks as Function Primitives: Software/Hardware Support with X-FILES/DANA," Boston Area Architecture Workshop (BARC) 2016

Whitepapers

- H. Dong, S. Arora, Y. Awad, T. Unger, O. Krieger, J. Appavoo, "Slowing Down for Performance and Energy: An OS-Centric Study in Network Driven Workloads," arXiv:2112.07010v1 [cs.OS] 13 Dec 2021

Talks

- T. Unger "Dynamic Privilege: A Thesis Defense," Boston University, Boston, USA, Dec 15, 2023
- T. Unger "Unikernel Linux (UKL)," Virtual talk to vHive community meeting, Nanyang Technological University (NTU) Singapore, July 4, 2023
- T. Unger "Unikernel Linux (UKL)," EuroSys Conference 2023, Thu, 11 May, Rome, Italy
- T. Unger "Building Flexible Systems with Just-in-Time Privilege," Boston University, Sep 26, 2023
- T. Unger "OS Extension from the Application Perspective," Boston University, Sep 12, 2022
- T. Unger "New Sheriff in Town: Deputizing Application Threads to Kernel Privilege for Performance, Profit, ... and Safety???" Red Hat DevConf, Aug 18 2022
- T. Unger, "Abstraction, Programmability, & Optimization: The Sandbox and the Sanctuary," BU Systems Seminar, May 6 2022
- A. Raza, T. Unger "Unikernel Linux," Red Hat DevConf, Sep 3 2021
- A. Raza, T. Unger "Unikernels," Red Hat DevConf, Sep 23 2020
- T. Unger "Optimizing Operating Systems," Red Hat Summit 2019
- T. Unger "FaaS: Think Outside the Container," MOC Day Workshop, 2018
- T.Unger "New Directions in Forensic Anthropology: Machine Learning Techniques for Sex Estimation," Forensic Sciences Symposium, BU School of Medicine, speaker April 10, 2015

Posters

- "Introducing the Chrono-kernel: Kernel Privilege for the People," T. Unger, A. Albelli, O. Krieger, J. Appavoo, L, Woodman, U. Drepper, R. Jones, D. Bristot de Olivera(). Research Seminar, Red Hat Research, 2023. Affiliations: Boston University, Red Hat. Research supported by Red Hat.
- "Unikernel Linux (UKL)," A. Raza, T. Unger, M. Boyd, E.B. Munson, P. Sohal, U. Drepper, R. Jones, D.B. de Oliveira, L. Woodman, R. Mancuso, J. Appavoo, O. Krieger Poster presented at Red Hat's DevConf, 2023. Affiliations: Boston University, MIT CSAIL, Red Hat. Research supported by the Red Hat Collaboratory at Boston University and the NSF.
- "X-FILES/DANA: RISC-V Hardware/Software for Neural Networks," S. Eldridge, Dong, H., Unger, T., Sahaya Louis, M., Delshad Tehrani, L., Appavoo, J., and Joshi A. Fourth RISC-V Workshop 2016.
- "Learning-on-chip using Fixed Point Arithmetic for Neural Network Accelerators," S. Eldridge, Sahaya Louis, M., Unger, T., Appavoo, J., and Joshi, A. Design Automation Conference (DAC) 2016
- "ASC: Automatically Scalable Computation - A Bridge between Worlds," A. Waterland, Y. Wang, T. Unger, T. Petrovic, S. Eldridge M. Seltzer, R. Adams, D. Brooks, J. Appavoo, S. Homer, A. Joshi. Exploiting Parallelism and Scalability (XPS), NSF. Annual report 2016

Affiliation

- Programmable Smart Machines Lab (PSML) Research Group BU 2014 - pres.
- Scalable Elastic Systems Architecture (SESA) Research Group BU 2017 - pres.
- MuSun Precision Particle Physics Experiment BU 2012-2013

Technical Skills, Other

Languages: Proficient: Python, C/C++, Bash, Makefiles, Assembly. Familiar: Rust, Go, Javascript
Codebases/Build Systems: Linux, glibc, GCC build system.
Tools: Jupyter Notebooks, GCC/Clang compilers, Perf, e-BPF. Git, Docker, VS Code.
Libraries: NumPy Seaborn, PyTorch, Fast Artificial Neural Network Library.
Activities: Founder of Math/CS Weekly Meditation Practice Group 2019-present.
Awards: Best Teaching Fellow 2018.