BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

KERNEL AS A SHARED OBJECT: DYNAMIC PRIVILEGE WITH DYNAMIC LINKING

by

FREDRIK L. J. WILKE

B.Sc., Hamburg University of Technology, 2025

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2026

Approved by

First Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

Second Reader

Jonathan Appavoo, PhD
Associate Professor of Computer Science

Third Reader

Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

Success consists of going from failure to failure without loss of enthusiasm.
Winston Churchill

Acknowledgments

I would like to thank everyone who supported me during the process of writing this thesis. First and foremost, I would like to thank Jonathan Appavoo for his guidance in every aspect of this work. I am grateful for everything I have learned working with him and for his support during the process of writing this thesis. I would also like to thank Orran Krieger and Manuel Egele for their continuous feedback and support during this project. Further, I would like to thank Prof. Appavoo's PhD students, Arlo Albelli, Eric Munson, Yara Awad and CJ Parra for their help and support and for welcoming me into their community. Finally, I would like to thank my family, friends and beloved Alika for their support and encouragement during this process.

KERNEL AS A SHARED OBJECT: DYNAMIC PRIVILEGE WITH DYNAMIC LINKING

FREDRIK L. J. WILKE

ABSTRACT

Uni-kernels have demonstrated that Kcuts, short-cuts from application to kernel code, are a powerful optimization technique in the construction of systems dedicated to a single task. Previous work has suggested that the power of such short-cuts can be naturally extended to a general purpose Operating System (OS) with modest effort. Support for Kcuts within an existing general purpose OS opens the door to a new approach for constructing optimized dedicated runtimes. Rather than conforming to the constraints of a uni-kernel, one can construct a runtime from an unrestricted set of user software while having the freedom to develop and exploit Kcuts for the task-critical binaries and libraries. The design focuses on making Kcuts an additive, optional, execution-time property that can be easily added to an OS kernel, in contrast to being a systemic property that impacts the complete design and implementation of the kernel and user software. In line with this, the OS side implementation only adds a new system call and a small attendant set of supporting kernel changes. The user-side implementation that allows applications to utilise kernel functions and data structures is the focus of this work. It presents a novel application and linkage model that allows for efficient application development and portability. This enables user binaries and libraries to be developed to enact particular kcut-based optimizations while integrating and exploiting the rich existing ecosystem of the OS. Overall, this work demonstrates Kcut optimization can enable functionality akin to uni-kernel's and yet be a standard feature of a general purpose OS.

Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface

CDF Cumulative Distribution Function

DPDK Data Plane Development Kit

DPLD Dynamic Privileged Load-Linker

eBPF extended Berkeley Packet Filter

EFB Exception Frame Buffer

ELF Executable and Linkable Format

GOT Global Offset Table

HBB Hybrid Build and Binary

HPC High Performance Computing

IDT Interrupt Descriptor Table

IST Interrupt Stack Table

KASLR Kernel Address Space Layout Randomization

libOS Library Operating System

MSR Model-Specific Register

NIC Network Interface Card

OS Operating System

PLT Procedure Linkage Table

RDMA Remote Direct Memory Access

SSD Solid State Drive

TSS Task State Segment

UKL Unikernel Linux

XDP eXpress Data Path

Chapter 1

Introduction

This thesis builds on two prior research efforts: Firstly, Unikernel Linux (UKL) and secondly, Dynamic Privilege (Raza et al., 2023)(Unger, 2024). Specifically, it explores whether it is possible to use the source and binary of an OS Kernel as if it were a standard body of reusable source code and a dynamic shared library to build "privileged" applications.

UKL demonstrated that opportunities exist for significant performance improvements by calling interior kernel routines directly from within the hot paths of an application — short-circuiting functionality not required given the dedicated high-performance use case being targeted. However, to enable this ability, UKL required a completely restructured application build environment and runtime (see 1.1). Dynamic Privilege demonstrated that with minor modifications, a general-purpose OS can support credentialed processes that can "elevate" and "lower" their hardware privilege on demand. Specifically, a user binary can be constructed that uses a new system call, Elevate, to execute with the same hardware privilege as the OS kernel.

This permits the construction of new application build and runtime models that exploit hardware privilege not only to execute privileged instructions but also to access the internals of the OS kernel. This thesis explores the construction of a hybrid application that makes it easy to combine both standard application code and libraries with the source and binary of an existing OS kernel. The goal is an

environment in which the shortcutting optimizations of UKL can be brought to the mainstream in a way that is familiar and normative for application developers. The contributions of this thesis are as follows:

1. Kernel as a Library (libKern.so): A new approach to exploiting the binary of the kernel as a standard user library that static and dynamic linkers can resolve application references to (this includes an attendant load methodology that exploits Dynamic Privilege).

2. Hybrid Build and Binary (HBB): A new "fat binary" for hybrid applications, which contains both binary objects to be loaded by the standard user dynamic link loader and binary objects to be link loaded by the existing kernel dynamic link loader. (This includes a build environment that makes it easy for an application developer to "extend" their application with code that reuses OS kernel source and headers).

3. A runtime for exploring Kernel Short-cuts (Kcuts): Using 1 and 2, the design and implementation of a new set of Dynamically Privileged extensions that enable a runtime in which four hybrid models of execution can be explored to enact short cuts from an application to interior call sites of the existing kernel. The four models are: 1) Bracketed Privilege & Bracketed Stack (BPBS), 2) Constant Privilege & Bracketed Stack (CPBS), 3) Bracketed Privilege & Constant Stack (BPCS), and 4) Constant Privilege & Constant Stack (CPCS).

The remainder of this thesis is organized as follows:

- Chapter 2: Implementation of libKern.so, HBB and Kcuts
- Chapter 3: Exploring Kcuts
- Chapter 4: Future Work

- Chapter 5: Conclusion.

1.1 Background and Related Work

In this section we briefly discuss the background and related work for this thesis. We begin by introducing Dynamic Privilege as an extension of UKL and review the context of these two research efforts. Then, we provide background information on different linkage models in kernel and user space and their implications for the design of libKern.so.

1.1.1 Dynamic Privilege

Dynamic privilege is a concept that is rooted in specialized Operating System (OS) designs such as Exokernels and Unikernels, that redefine the boundary between user applications and the OS to achieve performance improvements. The original work by Unger

> 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.

(Unger, 2024)

Unger cites Raza et al.'s Unikernel Linux as a primary influence that demonstrates that not only specialized library OSs can be used as the basis of a unikernel but that the Linux kernel itself can be used as a library OS(Raza et al., 2023). Dynamic

Privilege takes this idea further by giving applications the option to execute only selected parts of their logic in kernel mode instead of the whole application.

Implementation

Dynamic privilege is implemented as a small patch to the Linux kernel, composed of 173 lines of code across 5 files for the x86_64 architecture(Unger, 2024). The patch is centered around a new system call, `elevate`, that allows applications to obtain kernel privilege. It also features extensions to the `task_struct` to track the state of applications with respect to dynamic privilege and modifications to kernel exit path as well as the migration logic of processes between CPU cores.

The key modification among the above is the change to the kernel exit path. This code path is invoked whenever the kernel is about to perform a context switch to a user process. The modification checks if the process to switch to is a privileged process and if so, it performs a regular `ret` to the last program counter of the process instead of a `sysret` or `iret` which would switch from kernel mode to user mode. This allows a process to permanently run in kernel mode after it has called `elevate` even if it performs regular system calls while in kernel mode as the modification to the exit path applies to all system calls.

Related Work

Dynamic privilege was preceded by a long line of research on redesigning the boundary between user applications and the OS to achieve performance improvements.

Figure 1·1 extracted from (Unger, 2024) illustrates the broad range of work and mechanisms that relate to and influenced the development of Dynamic Privilege and UKL. A review of this entire body of work is beyond the scope of this thesis. We

refer an interested reader to Unger and Raza's theses.

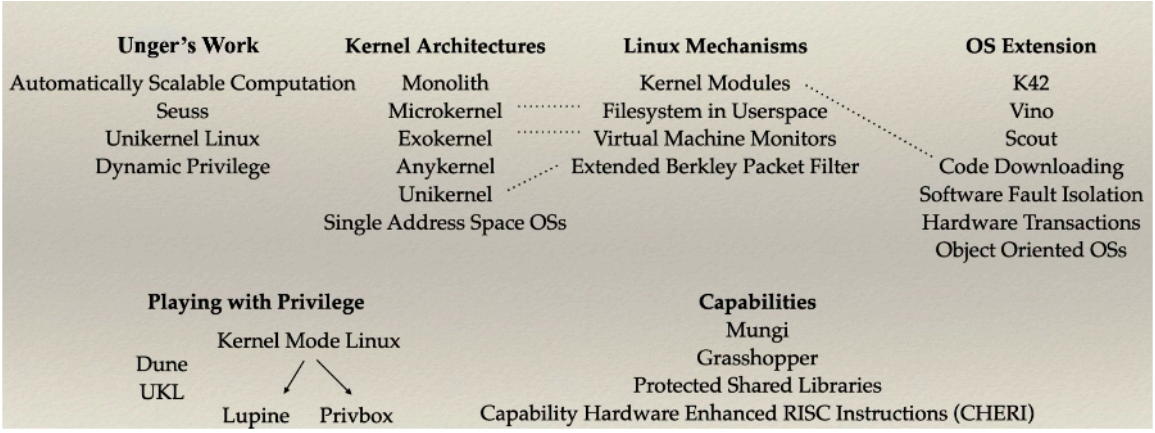


Figure 1·1: Image from (Unger, 2024) that illustrates the work and mechanisms that relate to and influence Dynamic Privilege.

Below we highlight two groups of related work that represent two key approaches to redesigning the boundary between user applications and the OS. Firstly, we present Exokernels and Unikernels as examples of foundational work within library OSs. Library OSs typically redesign the entire OS architecture to allow applications to access hardware resources directly. Secondly, we present kernel bypass frameworks as examples of a more incremental approach. These frameworks typically integrate with a general-purpose OS and allow applications to perform selected tasks that are usually performed by the OS without modifying the architecture of the OS.

Library OSs An Exokernel achieves the redesign of kernel boundaries by delegating resources available to the host machine to one or multiple applications (Engler et al., 1995). The applications run in user mode and make use of a library OS to manage the resources allocated to them. The library OSs interact with the Exokernel through a narrow low-level interface. The Exokernel maintains authority over the applications and implements mechanisms to prevent applications from performing unauthorized actions. We particularly draw out Exokernel as it is widely cited as

the origin for the concept of a library OS. The Exokernel work advocated that an application includes an application-specific library of OS functionality designed and implemented to uniquely and solely cater to the needs of the application.

Madhavapeddy et al, in "Unikernels: The Rise of the Virtual Library Operating System", builds on the Exokernel notion of a library OS (Madhavapeddy and Scott, 2014). In particular they coin the term Unikernel as a system that combined with a library OS allows a single application to run as a virtual machine on a hypervisor. This obliterates the distinction between user space and kernel space and context switches between the two.

UKL falls into this category with the unique aspect that it uses the Linux kernel as a library OS. This allows applications to make use of the large amount of functionality implemented in the Linux kernel and maintained by its large community while still achieving performance improvements by eliminating user-kernel boundary crossings (Raza et al., 2023). However, this approach comes with several limitations. Firstly, the application build environment changes significantly. UKL does not support dynamic linking of user space libraries and instead requires an application to be statically linked to all the libraries it uses. Furthermore, these libraries need to be recompiled with several options to ensure that they can be linked to the Linux kernel as part of the application binary. This complex build process is a significant barrier to entry and makes optimizing applications for UKL very time consuming. Secondly, UKL requires significant modifications to the Linux kernel and to `glibc`. It's patch size for the Linux kernel is up to 1,250 lines of code and 1,720 lines of code for `glibc`. Lastly, UKL requires a host to be dedicated to a single application and Unger suggested that Dynamic Privilege might offer an approach to achieving Uni-kernel advanages without the burden of entirely new OS configuraiton and application build

process without having to resort to a Kernel bypass framework.

Kernel bypass frameworks Kernel bypass frameworks are a different approach to redesigning the boundary between user applications and the OS. They don't modify the architecture of the operating system but allow applications to perform selected tasks that are usually performed by the OS. Prominent examples are eXpress Data Path (XDP)(Vieira et al., 2020), Data Plane Development Kit (DPDK)(Foundation,), and Remote Direct Memory Access (RDMA) which all redefine the process of handling network traffic. Given the industrial impact and investment in these by-pass framework we provide a brief descritpion below.

XDP allows users to specify packet handling rules using extended Berkeley Packet Filter (eBPF). The eBPF code is executed in the kernel context on arrival of an applicable packet. This eliminates the context switch that would usually occur if the packet was to be handled by a user space application. Similar systems have been developed that use eBPF to define scheduling algorithms or page cache management policies(kernel development community, ; Zussman et al., 2025). DPDK follows an orthogonal approach. It re-implements the kernel network stack in user space so that no context switches are required. However, a DPDK application needs to run on a dedicated CPU core and needs to granted exclusive access to a Network Interface Card (NIC). RDMA also eliminates the kernel from the dataplane of the application. However, it is not generally applicable as it requires specialized NICs that support the RDMA protocol. Consequently, its relevance is limited to server-to-server scenarios and High Performance Computing (HPC).

1.2 Building on Dynamic Privilege

In this section, we discuss how the work of this thesis extends and builds upon Dynamic Privilege to improve its usability and ultimately help realize the goal of making Uni-kernel optimizations easy to enact.

1.2.1 Linkage in Kernel- and User space

A key limitation of the Dynamic Privilege runtime prior to this research was its linking mechanism to the Linux kernel. It did not include a representation of the Linux kernel's symbol table in user space that could be used by user space linkers and loaders to link applications to the Linux kernel. Instead, Linux' symbol table as available in the `/boot/System.map` file was used to resolve symbols at runtime. This approach has several limitations.

- The addresses of kernel symbols can change between reboots due to Kernel Address Space Layout Randomization (KASLR). KASLR needs to be disabled to use `System.map` to resolve symbols.
- Symbols from loadable kernel modules are not included in `System.map` and thus cannot be resolved.
- The programmer or compiler toolchain needs to include code for each kernel symbol used to resolve it at runtime using `System.map`.

To mitigate these limitations, this thesis introduces and validates the ability to expose the binary representation of the running kernel as a standard dynamic shared library. We refer to this notion as libkern.so. In the current implementation, the name of the library is `libkallsyms` in acknowledgment of the existing kernel interface to its runtime symbols `/proc/kallsyms`.

`libkallsyms` is a library that is available in user space and exposes the Linux kernel's symbol table to user space linkers and loaders. User space libraries can be statically or dynamically linked to applications. Both approaches are supported by `libkallsyms` and are further elaborated in Chapter 2.

In the following, we provide background information on linkage methods in user space and kernel space that are foundational to the design of `libkallsyms`.

Static linkage

Static linkage refers to the process of linking an application to a library at compile time. The library typically comes as a `.a` file which is an archive of object files. It uses the `ELF` format to store the code and data of the library in a way that is readable by the linker. After the application is compiled and linked to the library, it can be executed without the need for the library to be present on the system. The library code is included in the application binary and is loaded into memory when the application is executed.

Dynamic linkage

Dynamic linkage refers to the process of linking an application to a library at runtime. The library typically comes as a `.so` file which is a shared object file. It also uses the `ELF` format to store its code and data. When using dynamic linking, the application binary does not include the library code. Instead, at compile time the linker includes references to the library symbols in the application binary. When the application is executed, the dynamic linker loads the library into memory and resolves the symbol references to the library code.

To facilitate this process, several data structures are included in the application binary. Most importantly, a Global Offset Table (GOT) is added to the application binary. The GOT is an array of addresses and is used to store the addresses of library

symbols that are resolved at runtime.

At compile time, the linker adds code to the application binary that allows the application to use the GOT to call library functions or access library data. For function calls, the linker adds an entry to the Procedure Linkage Table (PLT) for each library function used by the application. A PLT entry is a small piece of code that performs an indirect jump to the address of the library function stored in the GOT. Furthermore, PLT entries allow library symbols to be resolved lazily. When a PLT entry is called for the first time, it's GOT entry points to a resolver trampoline within the same PLT entry. This trampoline pushes the index of the PLT entry onto the stack and jumps to a common resolver function in the dynamic linker. The resolver function uses the index to determine which symbol needs to be resolved and stores its address in the GOT. Subsequent calls to the same PLT entry will then jump directly to the library function without the need for further symbol resolution.

For data symbols exposed by the library, the process differs. The linker either emits a copy relocation or adds an entry to the GOT with a corresponding relocation for each data symbol used by the application.

A copy relocation is a type of relocation that is used for data symbols. The linker-loader interprets it as an instruction to copy the value of the data symbol from the library into the application `.bss` section at application startup.

By default, a copy relocation is chosen for data symbols. This is because the default memory model used by user space applications on x86_64 is the small memory model which assumes that all code and data can be accessed using 32 bit relative addresses. Thus, also for symbols declared as `extern`, the compiler generates code that directly accesses them using 32 bit relative addresses. It emits a relocation that instructs the linker to fill in the relative address to the data symbol at link time. If

the symbol is defined in a shared object, then there is no guarantee that the symbol will be located within 32 bits of the application code. Also, the dynamic linker-loader would have to patch the application code at runtime to fill in the correct relative address. This would require the memory holding the application code to be writable which is a security risk. Thus, the linker chooses to reserve memory for the data symbol in the application `.bss` section and makes the application code point to this memory location. At application startup, the dynamic linker-loader copies the value of the data symbol from the library into the application `.bss` section.

For a GOT entry to be used instead, two conditions need to be met. The application code accessing the data symbol needs to be compiled as position independent code and with the large memory model. Only under these conditions, the compiler generates code for symbols declared as `extern` that accesses them through the GOT. Consequently, the linker chooses to add an entry to the GOT for the data symbol instead of a copy relocation. At application startup, the dynamic linker-loader then resolves the data symbol and stores its address in the GOT.

Kernel modules

With kernel modules, the Linux kernel also supports dynamic linking. A kernel module is also an `ELF` file that contains code and data. It can be loaded into the kernel at runtime using the `init_module` or `finit_module` system call.

Kernel modules are compiled using the kernel build system and headers and can reference a set of exported kernel symbols. Kernel symbols are not exported for use in modules by default. To make a kernel symbol available for use in modules, it needs to be exported using the `EXPORT_SYMBOL` macro in the kernel source code. As a consequence to kernel modules being compiled using the kernel build system, they in-

herit the kernel's memory model and thus rely heavily on the use of 32 bit pc-relative addresses. This means that kernel modules can only be loaded within the kernel's address space beyond `0xffffffff00000000` and within 32 bits of the kernel code.

When a kernel module is loaded, the kernel's module loader performs the necessary relocations to resolve the kernel symbols used by the module. Contrary to dynamic linking in user space, no PLT or GOT is populated in this process. Instead, the kernel's module loader directly patches the module code to fill in the offsets to the kernel symbols given the module's load address. This process is similar to the process of static linking in user space.

Chapter 2

Implementation

This section will present implementation details of the Kcut runtime and toolchain. Firstly it will describe the novel application model that Kcut uses, which is comprised of a user-space component and a kernel-space component. Secondly, it presents the design and implementation `libkallsyms`, a new library that enables privileged executables to dynamically link against kernel symbols. Finally, it will describe the modifications necessary to the existing dynamic privilege environment to improve stability and performance.

2.1 Kcut Application Model

The Kcut application model was designed with the following goals in mind:

- **Easy development**: The application model should allow developers to use familiar tools and workflows to develop Kcut applications.
- **Simple runtime**: The runtime of Kcut applications should not differ significantly from traditional applications.

To achieve these goals, Kcut applications were designed to consist of two components: a regular user-space executable and a kernel-space extension. The user-space executable is a regular ELF executable that is loaded by `ld.so` and executed in user mode. The kernel-space extension is a kernel module that is loaded into the kernel memory by Dynamic Privileged Load-Linker (DPLD) through the Linux kernel's

module installation mechanism. Figure 2·1 demonstrates the dual nature of Kcut applications and the loading process.

With this application model, developers can use any toolchain to produce their user-space executable while using the Linux kernel's native build system to produce their kernel extension. This avoids two key issues that our system would encounter if not using the kernel's module loader: memory model conflicts and symbol name conflicts.

Memory model conflicts The Linux kernel on x86_64 assumes that all kernel code and data structures are addressable using 32-bit relative offsets. For functions, this does not cause a problem when linking kernel binaries together with a user executable and `libkallsyms.so` because the linker adds PLT stubs for all external symbols. These stubs can be placed close to the kernel code that needs to call them, so the 32-bit relative offset is sufficient. However, for data symbols, a similar approach cannot be used. If GCC's linker encounters a 32-bit relative reloaction for a data symbol that is provided by a shared object, it will produce a dynamic copy relocation. This means that at load time, the dynamic linker will copy the data symbol from the shared object into the executable's memory and patch all references to it to point to the new location. This is highly undersirable for Kcut applications because it would not allow the application to access the kernel's data structures directly.

Symbol name conflicts Numerous symbols are defined in both the kernel and typical user-space libraries (e.g., `memcpy`, `strlen`, etc.). For a Kcut application, it needs to be ensured that no kernel symbols are accessed when the application is executing without kernel privileges as it would cause a segmentation fault. This implies that the linker needs to resolve some references to `memcpy` to the kernel's implementation and some to the user-space library's implementation. In such a scenario symbol versioning can be used to inform the dynamic linker-loader which symbol from which library

to use. However, this approach requires to manually specify the symbol version for each symbol that is defined in both the kernel and user-space libraries, which is not a scalable solution.

Embedding a kernel module as the Kcut application's kernel extension allows developers to avoid these conflicts. Using the kernel's runtime linker-loader alleviates the memory model conflicts as the kernel can load the module at an appropriate address and patch all references to kernel symbols. Also, the kernel's module loading mechanism allows the kernel to resolve all symbol references in the kernel extension to the kernel's symbols. The user-space executable can then resolve references to duplicate symbols to the user-space library's implementation.

Our application model requires custom logic to dynamically link the user-space executable and the kernel extension together after they are loaded into memory. In subsections 2.1.1 we discuss the various linkage scenarios supported by our HBB model.

To simplify the use of our HBB we implemented a prototype toolchain which is described in subsection 2.1.2.

2.1.1 Linkage between application components

The most typical way for the user-space executable to interact with the kernel extension is through function calls into the extension. Section 3.1 will provide examples of this. Similarly, the kernel extension may need to call functions in the user-space executable. For both cases, the application components need to be dynamically linked together after both components are loaded into memory at their final positions in memory.

User-executable to kernel-extension linkage: The kernel extension is loaded as a kernel module, by the kernel's module loading mechanism. This mechanism is invoked when ld.so resolves the address of one of the symbols contained in the

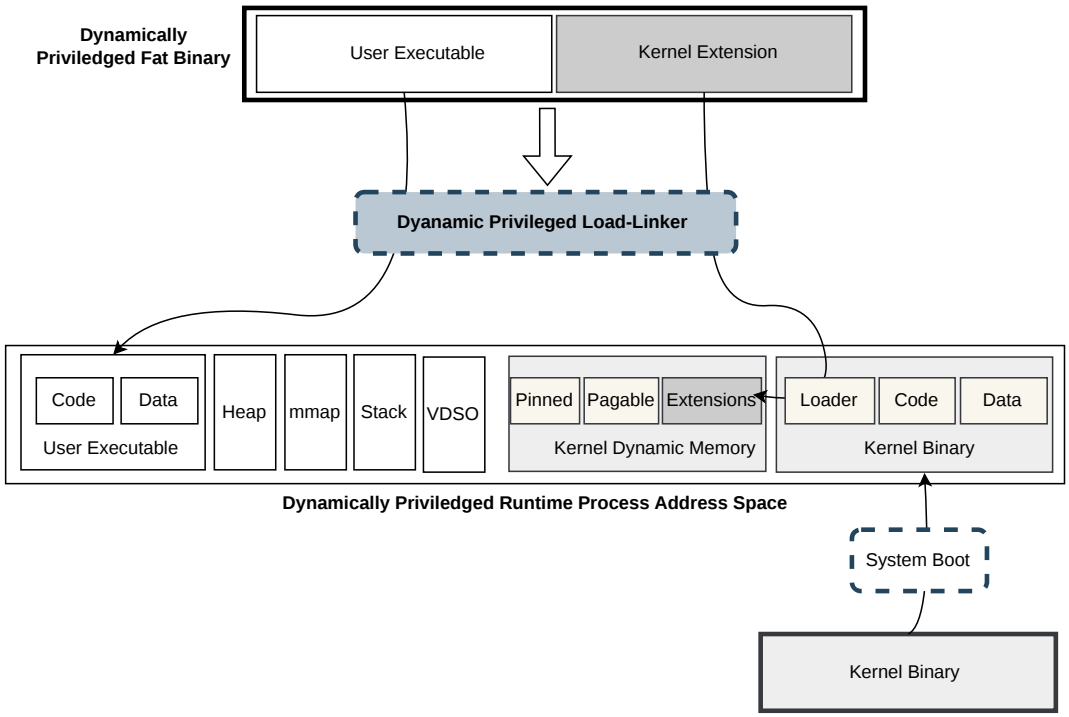


Figure 2·1: Kcut application model. A Kcut application consists of a user executable and a kernel extension. The dynamic privileged load-linker is triggered by ld.so at load time to load the kernel extension as a kernel module. Then, it dynamically resolves kernel and user symbols and patches symbol offset tables both the user application and the kernel extension.

kernel extension. This is implemented by using `ifuncs`, a libc feature that allows developers to specify a function that returns the runtime address of a symbol. Once the module is loaded, the kernel's own `kallsyms_lookup_name` function is used to resolve the address of the kernel extension's symbols. This address is then patched into the user-space executable's GOT.

Kernel-extension to user-executable linkage: For this scenario, the Kcut toolchain generates a GOT and a PLT in which the addresses of the user-space executable's symbols are stored. Once the kernel extension is loaded, the user-space executable fills the GOT with the addresses of the user-space symbols.

2.1.2 Kcut toolchain

The goal to provide an easy development experience for Kcut applications made it imperative be able to use kernel header files for the development of Kcuts applications. Especially with the kernel's large configuration space, it is desirable to be able to use the kernel's build system as data structures and the availability of certain symbols can change based on the kernel configuration.
Guided by this goal, the Kcut toolchain was designed to integrate artifacts built by the kernel's build system into the user-space executable. This is presented in Figure 2·2. However, the toolchain also aims to not segregate the user-space source code from the kernel-space source code to not cause unnecessary fragmentation of the codebase.

We adopt a "compound executable" or "fat-binary" model (Thompson, 1989; NeXT Computer, Inc., 1993). Our HBB approach is similar to NVIDIA's. It uses unique file extensions and an associated toolchain to provide a simple model for developers to organize, author, and reuse hybrid source code and binaries.

The Kcut toolchain defines several new file extensions that are used to indicate the intended purpose of each source file.

- `.kc` indicates a c-source file that is intended to be compiled using the kernel's build system.
- `.kh` indicates a header file that is intended to be included in source files compiled by the kernel's build system.
- `.kS` indicates an assembly source file that is intended to be compiled using the kernel's build system.

The regular file extensions used for C and assembly source files and header files indicate that they are intended to be compiled using a regular user-space toolchain. To integrate the object files produced by the two different build systems, the Kcut

toolchain produces additional logic for both the user executable and the kernel extension. For the user executable, the Kcut toolchain produces the definitions of the kernel extension's global symbols as `ifuncs` as well as the logic to trigger the loading of the kernel module and to populate the kernel extension's GOT. For the kernel extension, the Kcut toolchain produces a GOT and a PLT to link to the user-space executable.

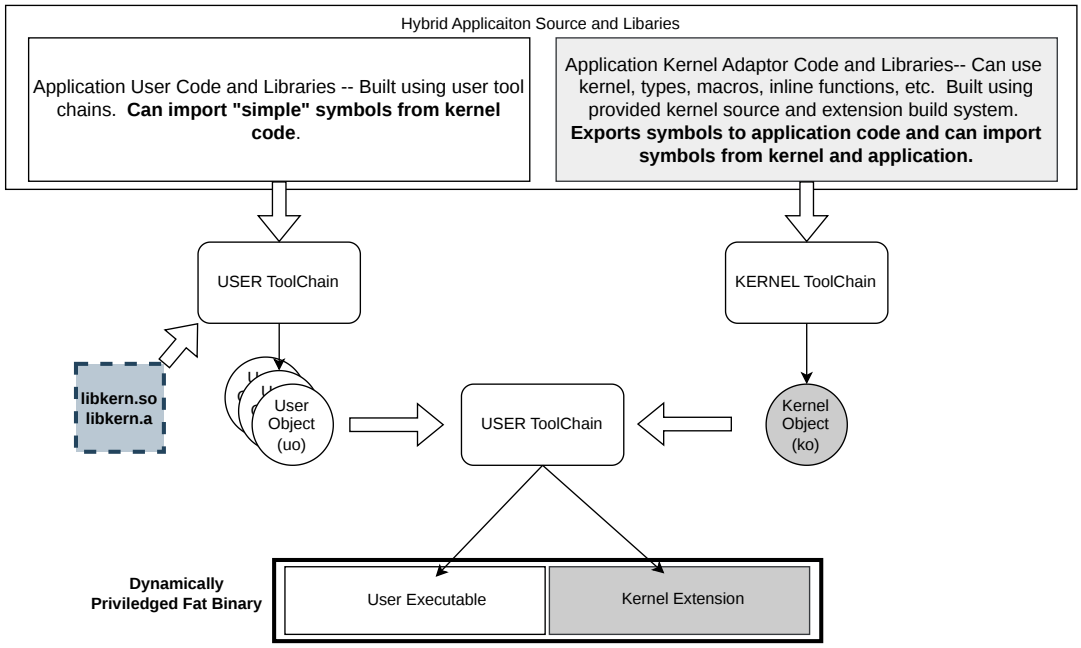


Figure 2·2: Kcut toolchain. The user toolchain is extended to include the applications kernel extension (.ko) into the final executable. This allows developers to use the Linux kernel's native build system for the kernel extension.

2.2 `libkallsyms` Design and Implementation

`libkallsyms` is a central component of the Kcut runtime that allows Kcut applications to dynamically link against kernel symbols. it provides the capability for Kcut applications to directly call kernel functions and access kernel data structures with-

out the need for a shim layer in the application's kernel extension. This can be used to implement "shallow shortcuts" to the kernel, which directly call into the kernel functions that implement certain system calls. These shortcuts have been shown to provide performance improvements for certain system calls(Unger, 2024).

2.2.1 Interfaces

Figure 2·3 presents the interfaces of `libkallsyms` and how it interacts with the kernel and user space.

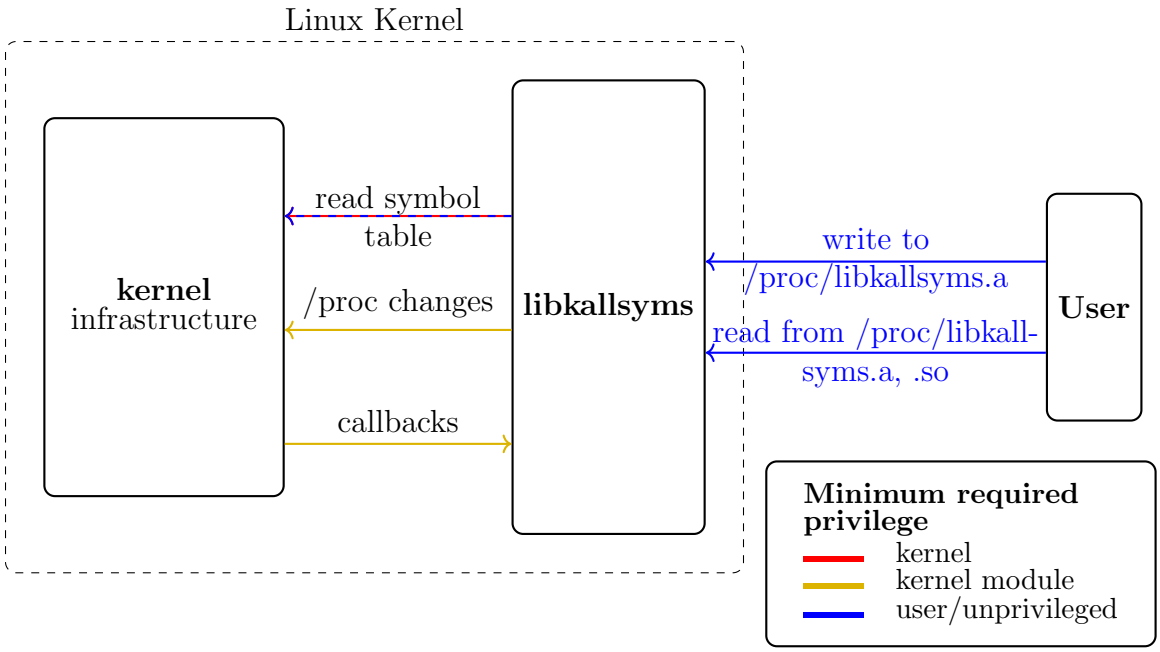


Figure 2·3: `libkallsyms` interfaces. `libkallsyms` interacts with the Linux kernel to read its symbol table and to register `libkallsyms.a` and `libkallsyms.so` in the /proc directory. Also, it uses the kernel's `register_module_notifier` interface to register a callback on kernel module load/unload. The user interacts with `libkallsyms` by reading and writing virtual files in the /proc directory. The different privilege levels required for each interaction dictate the placement of `libkallsyms` as a part of the kernel.

User interface

To the user, it provides two virtual files in the /proc directory: `/proc/libkallsyms.a` and `/proc/libkallsyms.so`. The former is a static library that exposes the kernel's symbols. It can be used to link the user-space executable against the kernel's symbols at compile time. This is useful when KASLR is not enabled and the kernel's symbols are at fixed addresses. The latter is a shared object that also exposes the kernel's symbols. It can be used to link the user-space executable against the kernel's symbols at compile time and to resolve the kernel's symbols at load time. This is useful when KASLR is enabled and the kernel's symbols are at random addresses. It also ensures safe resolution of symbols provided by kernel modules as it fails gracefully when a required module is not loaded.

Using `libkallsyms.a` requires the user to select the symbols that the application needs to link against at compile time. This is required because the kernel's symbols can prevent the application from linking against the user-space libraries' symbols due to symbol conflicts. This constraint only applies to `libkallsyms.a` because symbols provided by static libraries generally take precedence over symbols provided by shared objects. On the other hand, using `libkallsyms.so` can expose all kernel symbols to the user-space executable without causing symbol conflicts because the order of precedence between shared objects can be controlled by the user.

One caveat of our current prototype using virtual files in /proc is that these can not be mapped as executable memory. Consequently, `libkallsyms.so` needs to be copied to a regular file before it can be used at runtime. We leave it to future work to address this with a more complex synthetic filesystem implementation that supports executable mmaps.

Kernel interface

`libkallsyms`'s interaction with the kernel is threefold. Firstly, it needs to read the kernel's symbol table to produce `libkallsyms.a` and `libkallsyms.so`. Secondly it needs to register the virtual files in /proc and the callback handlers for interactions with these files. Finally, it needs to register a callback on kernel module load/unload to update the information provided by `libkallsyms.so` and `libkallsyms.a` when kernel modules are loaded and unloaded.

Each of these interactions requires a different level of privilege. The kernel's symbol table is only accessible from within the kernel. Kernel modules do not have access to the kernel's symbol table. However, authorized user-space applications can read the kernel's symbol table through the /proc/kallsyms virtual file. Registering the virtual files `libkallsyms.a` and `libkallsyms.so` in /proc is only possible from within the kernel or a kernel module. However, a user-space application could register these files in another filesystem (e.g., /tmp). Thus, a user-space application could implement a version of `libkallsyms` that only provides native kernel symbol information.

However, this is a less favorable option as Kcuts applications may need to link against symbols provided by kernel modules and registering a callback on kernel module load/unload is significantly easier from within the kernel or a kernel module.

This leads to the design choice of implementing `libkallsyms` as a translation unit of the Linux kernel.

2.2.2 Internal structure

The internal structure of `libkallsyms` is presented in Figure 2·4.

The main data structure used by `libkallsyms` is a linked list `lks_modules` of `lks_module_t` structures that store the symbol information of the kernel and all loaded kernel modules. Each element of the list contains a reference to the sym-

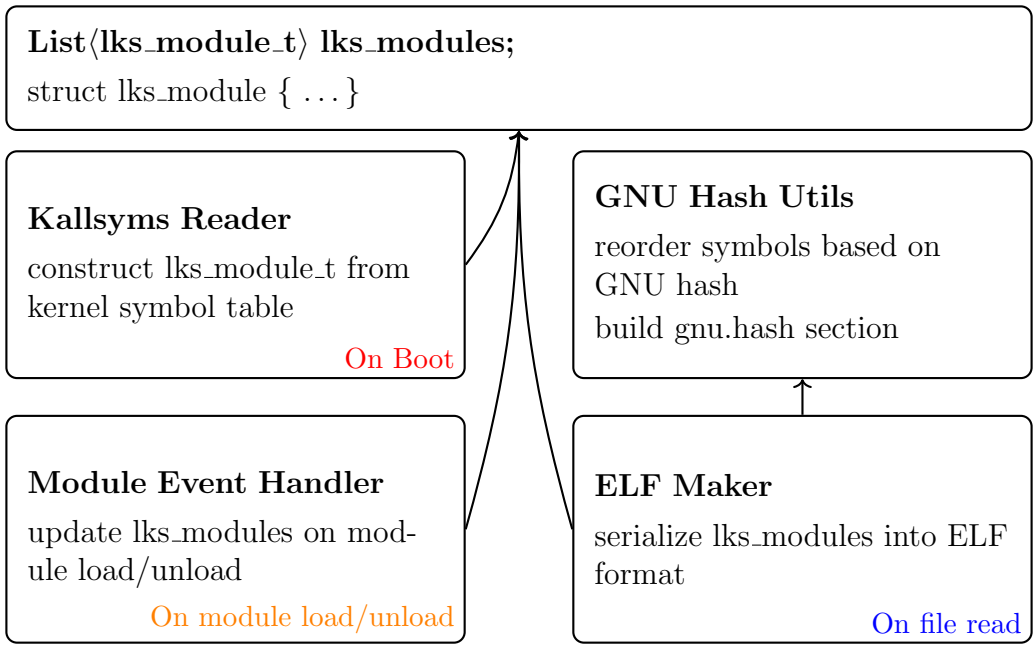


Figure 2·4: `libkallsyms` internal structure. A linked list of `lks_module_t` structures is maintained to store all kernel and kernel module symbol information. It is used to produce the ELF files that are exposed to user space. The GNU hash utilities are used to generate ELF sections required by ld.so for shared objects.

bol table and the string table of the kernel or a kernel module as well as their sizes. This data structure is initialized at boot time with the kernel's symbol table and is updated whenever a kernel module is loaded or unloaded.

When `libkallsyms.a` or `libkallsyms.so` is read, `lks_modules` is serialized into an Executable and Linkable Format (ELF) file. For the case of `libkallsyms.a`, the symbol and string tables of the kernel and all loaded kernel modules are simply concatenated to produce its symbol and string tables. For the case of `libkallsyms.so`, a `.gnu.hash` section needs to be generated to allow `ld.so` to quickly resolve symbols at load time. For this, the symbols need to be reordered based on their hash values

and a hash table needs to be generated based on the GNU hash algorithm.

2.3 Runtimes for Kernel Short Cuts

We made several modifications to the existing dynamic privilege environment to improve stability and performance(Unger, 2024) and enable the exploration of UKL inspired "deep-shortcuts".

1. We designed and implemented a new model for handling exceptions to ensure that a privileged application need not deal with inconsistencies of suffering exceptions and interrupts while it is executing.

2. The privilege transition mechanism was modified to ensure that the system is always in a consistent state with respect to the privilege level and the state of the `gs` register.

3. The logic to handle CPU migration of Kcut applications between cores was improved to prevent inconsistencies.

4. The system was forward-ported to Linux kernel 6.16 from the original implementation on Linux kernel 5.14.

2.3.1 Interrupt handling and stacks

A challenge that was faced by UKL and Dynamic Privilege is how to deal with interrupts and interactions with the processor's stack(Raza et al., 2023). On x86, there is a complex set of scenarios that challenge the construction of a hybrid runtime that intermixes application code execution and the kernel. As part of our effort to explore UKL-style deep shortcut optimizations, we have developed a novel approach to addressing these challenges. Specifically, we have developed a set of reusable kernel extension code that a Kcut application developer can reuse to explore four execution

models (See 3.1.1 for a discussion of these models). At the heart of our approach is a novel x86 implementation of an Exception Frame Buffer (EFB). By reprogramming the interrupt dispatching structures, we ensure that on any standard interrupt or exception, the hardware will write its Interrupt Stack Frame or Trap Frame onto a known per-core, pinned, and dedicated page of memory. This way, regardless of what was executing at the time of the interrupt or exception, a privileged process, an unprivileged process, or the kernel proper, the hardware will not disturb the state of the stack in operation. By then, running a newly developed common handler, we can determine the correct location for the hardware-pushed frame and transparently compensate for hybrid execution scenarios as needed. After this code executes, it will ensure that the original constraints and expectations of the standard kernel split-privilege model are restored, and the original handler for the exception or interrupt is invoked.

In this section, we review the implementation of this EFB mechanism on x86.

EFB using x86 Interrupt Stack (IST) support.

When an interrupt or exception occurs on x86, the CPU pushes an exception stack frame onto the current stack. The mechanism by which the hardware selects the location of this stack frame is shown in Figure 2·5. First, the CPU checks whether the Interrupt Descriptor Table (IDT) entry for the interrupt or exception defines an index of an Interrupt Stack Table (IST) stack. If it does, the CPU pushes the exception stack frame onto the specified IST stack. This method is used for critical exceptions such as double faults to ensure that the CPU can handle these exceptions even when the current stack is corrupted. However, Linux does not use IST stacks for regular interrupts and exceptions. We capitalize on the IST mechanism to add a level of EFB indirection.

If the IDT entry does not define an index of an IST stack, the CPU checks whether

the interrupt or exception occurred while the CPU was executing in kernel mode or user mode. If the CPU was executing in kernel mode, the exception stack frame is pushed onto the current stack. If the CPU was executing in user mode, the CPU switches to the kernel stack RSP0 defined in the currently active Task State Segment (TSS).

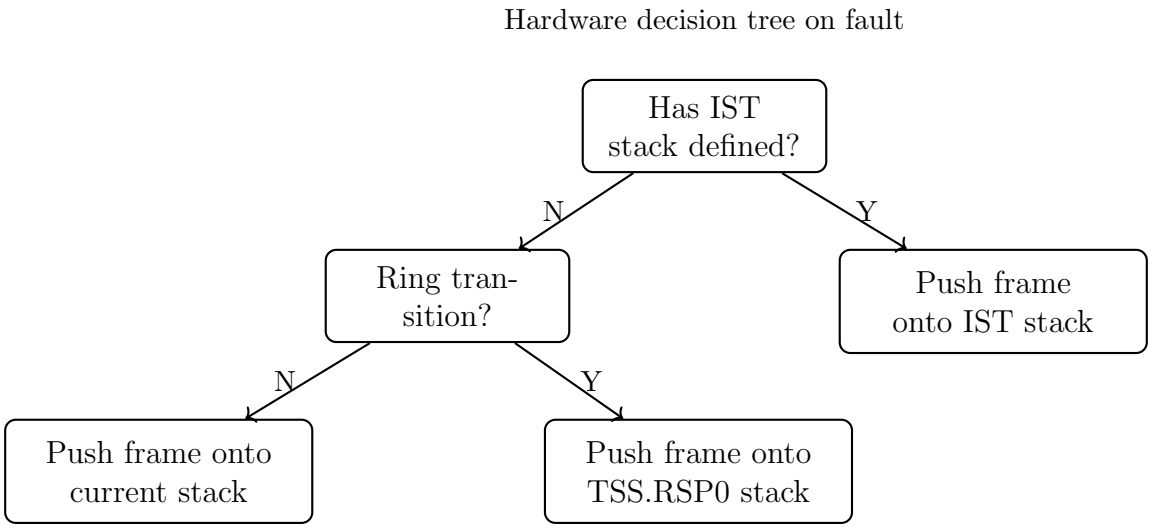


Figure 2·5: x86-64 hardware decision tree for interrupt stack frame placement. The exception stack frame is placed on the current stack if the CPU is already in kernel mode when the interrupt occurs. This can cause stack corruption when executing in kernel mode with a user stack.

This mechanism can lead to multiple vulnerabilities with respect to dynamic privilege.

Stack access vulnerabilities Firstly, if an application is executing in kernel mode with a user stack, then this stack will be reused for all interrupts and exceptions that do not have an IST stack defined. This can lead to fatal errors because the application's stack does not necessarily meet the kernel's expectations. Specifically,

application's stacks are pageable and can be swapped out to disk, while the kernel expects the stack to always be resident in memory. Thus, the interrupt handler might fault on the application's stack, which leads to a double fault. Furthermore, certain interrupts such as timer interrupts might call into the scheduler, which expects to be running on a stack that is mapped in every process's address space. If the scheduler is invoked on a stack that is only mapped in the current process's address space, then it will suffer a double fault after switching to another address space.

Stack corruption vulnerabilities Secondly, handling interrupts and exceptions on an application's stack can lead to stack corruption. This is because leaf functions in user space may use the "red zone" of the stack, which is a 128-byte area below the stack pointer that is reserved for use by leaf functions (see Figure 2·6). If an interrupt or exception occurs while the application is on an application's stack, in kernel mode and executing a leaf function, the CPU will write the exception frame into the red zone, corrupting the data used by the leaf function.

EFB Mitigation To mitigate these vulnerabilities, an EFB mechanism was implemented using IST stacks for all interrupts and exceptions. Figure 2·7 shows how this mechanism works. It uses the IST stack as a frame buffer to temporarily store the exception frame when an interrupt or exception occurs. Then, it copies the exception frame from the IST stack to the appropriate stack based on the stack pointer and the privilege level at the time of the interrupt. The following scenarios lead to the use of RSP0 stack:

- The interrupt occurs while the CPU is executing in user mode.
- The interrupt occurs while the CPU is executing in kernel mode, the stack pointer is in user space and the current task is marked as dynamically privileged.

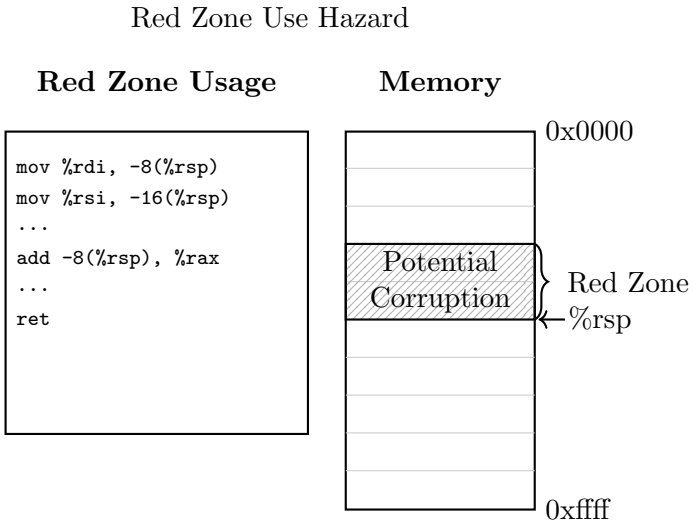
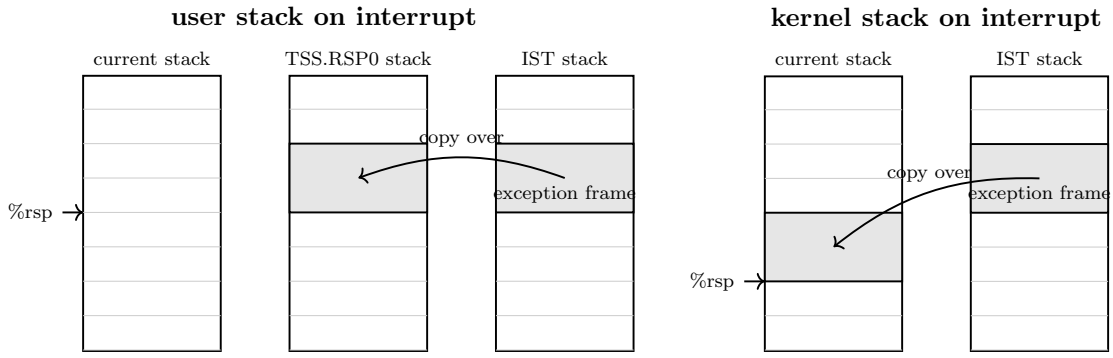


Figure 2·6: Red zone memory hazard in privilege transitions. Leaf functions may use the memory area below the stack pointer. If an interrupt occurs that pushes an exception frame onto the current stack, this data can be corrupted.

In all other cases, the exception frame is copied to the stack pointer at the time of the interrupt. This mechanism eliminates both stack access and stack corruption vulnerabilities as it ensures that interrupts and exceptions are always handled on a kernel stack and that the application's stack is never used for interrupt handling. Quantifying the performance impact of EFB is left to future work. However, it has been designed under the assumption that an application switches the system to using the EFB approach once the system is "hot" and the critical performance-sensitive paths typically do not induce exceptions or rely on a frequency of interrupts for which the hand-optimized assembly code will impact performance.

2.3.2 Privilege transition consistency

Another issue we encountered when attempting to implement UKL like deep short cuts were inconsistencies in critical register state when there is a lack of privilege transition between user and kernel code on interrupts/exceptions. The Linux kernel



(a) User mode when faulting

(b) Kernel mode on fault

Figure 2·7: Interrupt handling with IST interposition. Exception frames are copied from the IST stack to the stack appropriate stack based on the stack pointer at the time of the interrupt, not based on the privilege level.

makes assumptions about the state of the CPU's registers when entering an interrupt handler. Specifically, on x86-64 it makes assumptions about the state of the `gs` segment register based on the privilege level that the CPU was executing in at the time of the interrupt. Linux assumes that when the CPU was executing in kernel mode, the `gs` register is set the offset of this CPU's set of per-CPU data structures. This results in a requirement to the privilege transition mechanism of the Kcut runtime. It must ensure that the system does not encounter an interrupt or exception while the system is in an inconsistent state with respect to the privilege level and the state of the `gs` register. Figure 2·8 illustrates this requirement. To achieve idempotency of the privilege transitions the following mechanisms were implemented.

For the **elevate** transition, the system uses the `elevate` system call which returns with interrupts disabled. Then, the `gs` register is updated based on the `kernel-gs-base` Model-Specific Register (MSR) and are re-enabled. Furthermore, alignment requirements are used to ensure that the transition code does not cross a page boundary. For the **lower** transition, the system disables interrupts before updating the `gs` register

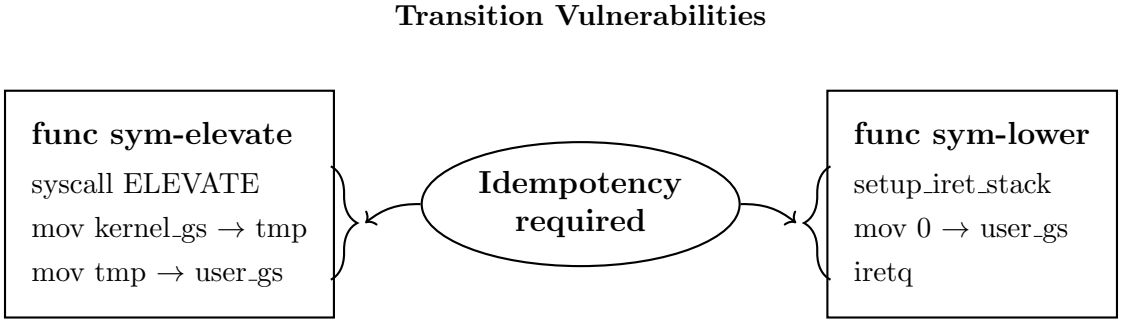


Figure 2·8: Privilege transitions require idempotency. Interrupts or faults ocurring when the system is in an inconsitent state with respect to privilege mode and the state of the gs register can lead to fatal errors.

to its user-space default value 0. Then, an `iretq` instruction is used to return to user space and to enable interrupts atomically. For this code alignment is also used to ensure that the transition code does not cross a page boundary.

Chapter 3

Exploring Kcuts

The main goal of this chapter is to demonstrate that the contributions of this thesis result in a stable and highly usable runtime and build environment for Kcut applications. For this purpose, a case study based on three different applications is conducted. The applications are of increasing complexity, starting with lebench, a microbenchmark suite that evaluates the latency of individual system calls, followed by Valkey, a single-threaded key-value store, and finally memcached, a multi-threaded key-value store.

This chapter first describes the case study design. It introduces the configurations considered, details about the applications, their modifications, and the experimental setup in which the case study was conducted. Then, the results obtained from the case study are presented. Overall, it is shown that the thesis work has now provided a foundation of functionality such that we are now in a position to get multi-threaded behavior debugged and begin the task of performance optimization for real world apps.

3.1 Case Study Design

The case study is designed to evaluate the usability of our HBB implementation, libKern.so and the stability of our Kcut runtime. It explores each of the applications behavior under 6 different configurations, which are chosen to illustrate the decision space provided by our runtime.

3.1.1 Configurations

For each application, the performance and stability was observed for six different configurations. These configurations explore the decision space provided by our runtime in two respects. Firstly, they explore the effect of "conformity", i.e., the extent execution "conforms" to the kernel's invariants and assumptions, or how closely the application resembles regular kernel execution when it calls into kernel code by either switching to a kernel stack or not. Secondly, they explore the effect of changes in privilege change frequency, i.e., how often the application transitions between user and kernel mode. This is done by either obtaining kernel privilege at launch and retaining it for the entire execution or by obtaining kernel privilege only when needed, i.e., right before calling into kernel methods. The case study does not explore the effect of calling into kernel code at different "depths", meaning how many function/system calls are bypassed by the optimizations compared to the original application.

The configurations are as follows:

Baseline: The application running without any optimizations on a standard Linux kernel.

Base-IST: The application running with the IST interposition feature running on a kernel compiled to support Kcut applications. This serves to quantify the overhead of the EFB interposition mechanism itself.

Kcut-CPCS (Constant Privilege, Constant Stack): This configuration is built on top of the Base-IST configuration. Constant privilege refers to the fact that the application obtains kernel privilege at launch and retains it for the entire execution. Constant stack means that the application does not switch to a separate kernel stack before calling into kernel methods, but instead stays on the user stack.

Kcut-CPBS (Constant Privilege, Bracketed Stack): This configuration is the same as Kcut-CPCS, but it switches to a separate kernel stack before calling into kernel

methods.

Kcut-BPCS (Bracketed Privilege, Constant Stack): This configuration is the same as Kcut-CPCS, but it obtains kernel privilege only when needed, i.e., right before calling into kernel methods.

Kcut-BPBS (Bracketed Privilege, Bracketed Stack): This configuration is a combination of Kcut-CPBS and Kcut-BPCS, i.e., it obtains kernel privilege only when needed and it switches to a separate kernel stack before calling into kernel methods.

Using these configurations is both helpful in determining stability issues of the runtime and in quantifying the impact of the different optimizations. It is expected that the Kcut-BPBS configuration will be most stable, as it causes the application to behave most similarly to its original version, both in application code and in kernel code. Meanwhile, the Kcut-CPCS configuration is expected to have the highest performance, as it should not incur any overhead from switching stacks or obtaining kernel privilege.

Lebench

Lebench is a microbenchmark suite introduced by Ren et al.(Ren et al., 2019) to evaluate the execution time of individual system calls. For this case study, only a subset of system calls is used, namely `read`, `write`, `send`, and `recv`. Each system call is executed in a loop for a fixed number of iterations, and the total execution time is measured using `clock_gettime`. Across the experiments, the amount of data transferred by the system calls is increased to evaluate the impact of the optimizations on different data sizes.

Listing 3.1 shows the core of the lebench read benchmark, demonstrating how the different configurations are implemented using preprocessor directives. The flag `BRACKET_PRIV` controls whether privilege is bracketed (elevated before and lowered

after the call), while `SYM_SHORTCUT` enables direct kernel calls instead of using the syscall interface. When `BRACKET_STACK` is defined, a stack switch is performed before calling the kernel function via a thunk. `ksys_read` is declared as an external function. `ld.so` resolves its address using libKern.so when it loads the application.

Listing 3.2 shows an alternative approach by Unger(Unger, 2024), which uses function pointers obtained at runtime to call kernel functions directly. `get_fn_address` uses `System.map` to obtain the address of the kernel function in this version.

Listing 3.1: Lebench read benchmark showing the different shortcutting configurations

```
#include "kernel.h" //declares ksys_read

int read_bench(){
    //...
    sym_elevate();
    clock_gettime(CLOCK_MONOTONIC, &runs[l].start);

    ksys_read(fd, buf, file_size);

    clock_gettime(CLOCK_MONOTONIC, &runs[l].end);
    symbi_fast_lower();

    //...
}
```

Listing 3.2: Previous approach to resolving kernel symbols in lebench under Dynamic Privilege (Unger, 2024)

```
typedef ssize_t (*read_t)(int fd, void *buf, size_t count);
read_t     sc_read;

void init() {
    //...
    sc_read = (read_t)get_fn_address("ksys_read");

    //...
```

```
}

int read_bench(){
    sym_elevate();
    clock_gettime(CLOCK_MONOTONIC, &runs[l].start);

    sc_read(fd, buf, file_size);

    clock_gettime(CLOCK_MONOTONIC, &runs[l].end);
    sym_lower();
}
```

Valkey

Valkey is a key-value store forked from Redis(contributors, b). It is a single-threaded application that serves TCP requests from clients over the network. It stores all data in memory and does not persist data to disk. Redis has previously been used by Raza et al.(Raza et al., 2023) and Unger(Unger, 2024) to evaluate their optimizations, making it a suitable choice for a comparison. Valkey is evaluated using the memtier benchmark(contributors, a), which is a standard benchmark for key-value stores and has been used in previous evaluations of Redis(Raza et al., 2023; Unger, 2024). The benchmark is configured with the following parameters:

- Number of clients: 50
- Number of threads: 6
- Total number of requests: 200000
- Set:Get ratio: 1:10 (default)
- Key size: 9 to 16 bytes (”memtier-0” to ”memtier-10000000”)
- Value size: 32 bytes

Latency is measured by the benchmark client and includes network latency.

Listing 3.3 shows the implementation of the socket write function in Valkey using kcut optimizations. The code demonstrates how Valkey's network I/O layer is modified to directly invoke kernel functions. `connSocketWrite` calls `kcut_tcp_write`, which in turn calls `kcut_tcp_sendmsg`, a kernel function that bypasses the syscall interface and directly invokes `tcp_sendmsg` from the TCP/IP stack. The kcut version translates the file descriptor to the underlying socket structure and initializes the necessary kernel data structures before calling the TCP send function. This implementation is an example of a deep shortcut optimization as originally explored by UKL(Raza et al., 2023). It illustrates that developers can easily write custom kernel functions that can be called directly from the application. A similar optimization was implemented for libc's `read`.

Listing 3.3: Valkey socket write implementation using kcut optimizations

```
//valkey/src/socket.c
#include "../../kcut/kcut_tcpmsg.h"

static int connSocketWrite(connection *conn, const void *data,
    size_t data_len) {
     //...
     int ret = kcut_tcp_write(conn->fd, (void *)data, data_len);
     //...
}


//kcut/kcut_tcpmsg.h
static inline
ssize_t kcut_tcp_write(int fd, void *buf, size_t count)
{
  struct iovec iov;
  iov.iov_base = (void *)buf;
  iov.iov_len = count;
  int ret;
```

```
  //CPCS version of kcut_tcp_write
  ret = kcut_tcp_sendmsg(fd, &iov);

  return ret;
}


//kcut/kcut_tcpmsg.kc
int kcut_tcp_sendmsg(int fd, struct iovec *iov)
{
  struct file *file;
  struct socket *sock;
  struct sock *sk;
  struct kiocb kiocb;
  struct msghdr msg;
  int ret = 0;

  //A: Translate fd to sock *
  //   From fs/read_write.c:ksys_write :
  //     1. fd -> struct fd :
  //        Fiven that fdget_pos is an internal routine we will use
     fget(fd) -- module friendly
  CLASS(fd, f)(fd); // acquire a 'struct fd' f from int fd -- clean
     when f goes out of scope - ref counted
  if (WARN_ON(fd_empty(f))) return ret;
  //     2. struct fd to file *
  file = fd_file(f);
  if (WARN_ON(!file)) return ret;

  //   3. file -> struct sock * : without reference counting since
     we have reference count on fd
  sock = file->private_data;
  sk = sock->sk;

  // B: from a combination of new_sync_write and sock_write_iter
     init kiocb and iov
  init_sync_kiocb(&kiocb, file);
  kiocb.ki_pos = 0;
  iov_iter_init(&msg.msg_iter, ITER_SOURCE, iov, 1,
                iov->iov_len);
  msg.msg_iocb = &kiocb;
```

```
  // C: send the message
  ret = tcp_sendmsg(sk, &msg, iov->iov_len);

  return ret;
}
```

memcached

Memcached is also a widely used in-memory key-value store. It is a multi-threaded application that serves TCP requests from clients over the network. As opposed to Valkey, memcached is by default multi-threaded. For the case study, memcached is configured to use 4 threads, and the memtier benchmark is used with the same parameters as for Valkey.

Listing 3.4 shows the implementation of the TCP write function in memcached using the same kcut optimizations as Valkey. The modification is equivalent to Valkey's implementation, calling `kcut_tcp_write` from its native `tcp_write` function. Memcached reuses the same `kcut_tcpmsg.h` header as Valkey and the same underlying implementation of `kcut_tcp_sendmsg`. Effectively, memcached and Valkey share the same kernel extension.

Listing 3.4: Memcached TCP write implementation using kcut optimizations

```
//memcached/memcached.c

#include "kcut_tcpmsg.h"

ssize_t tcp_write(conn *c, void *buf, size_t count) {
    assert (c != NULL);

    int ret = kcut_tcp_write(c->sfd, buf, count);
    return ret;
}
```

3.1.2 Experimental Setup

The experiments are conducted with the applications running on a bare metal server equipped with an Intel Xeon E5-2640v4 40 core CPU and 131 GB of RAM. The server is running either a standard Linux kernel or a kernel patched to include Kcut support, depending on the configuration. It does not run a standard Linux distribution, but instead a custom minimal initial ramdisk (initrd) image that includes only the necessary components to run the applications. With lebench, no client is needed to generate the workload. The benchmark is run directly on the server. For Valkey and memcached, the memtier benchmark is run on a separate host machine, which connects to the server over a network. Specifically, the server and client are connected via a 10 Gbps Ethernet link through a top-of-rack switch.

3.2 Stability Observations

Initially, the case study was performed without the adaptions made to the Kcut runtime as described in section 2.3. This led to several stability issues which were isolated and resolved. Table 3.1 summarizes the stability issues encountered during the case study and their resolutions. Most of the issues listed were so fundamental that they affected all applications and configurations and caused the system to suffer an unrecoverable fault.

One key issue that manifested in several ways was the modified setting in which interrupts and faults were handled prior to the introduction of the EFB based handling. One setting in which this caused issues was when a page fault occurred while handling another page fault in the kcut application code. A code example that provokes this issue is shown in listing 3.5. Here a page fault is caused by accessing `not_present_page2` while being on a stack to which only 2 more quadwords can be

Issue	Resolution	Applications affected
Page faults when handling faults in kcut application code	EFB based fault handling	lebench, Valkey, memcached
Faults when rescheduling after timer interrupts	EFB based timer interrupt handling	lebench, Valkey, memcached
Stack corruption by interrupts	EFB based timer interrupt handling	Valkey, memcached
Inconsistent state of `gs` on repeated privilege transitions	page align transition code, disable interrupts during transitions	lebench, Valkey, memcached
Inconsistent state of `gs` after process migration to another core	improved migration handling	lebench, Valkey, memcached
Rescheduling on I/O blocking	remove competing processes from core	lebench
Page faults when extending heap	none so far	memcached

Table 3.1: Stability issues encountered during the case study and their resolutions.

pushed. When the hardware tries to push the exception frame consisting of 6 quadwords to this stack, it causes a double fault, which is unrecoverable. Scenarios like this were probabilistically encountered during the case study, as it is generally safe for user space processes to fault on their user stack.

Listing 3.5: A code example that provokes a double fault when EFB based fault handling is not used

```
sym_elevate(); //get kernel privilege

new_stack_ptr = (char *)not_present_page + 16; //This
   leaves room for two more quadwords on the new stack

__asm__ volatile (
    "mov␣%0,␣%%rsp\n\t"
```

```
        :: "r" (new_stack_ptr) : "memory"
    );

    *(volatile char *)not_present_page2 = 0x99; //this causes
       a page fault
```

Other faults that were encountered that can be attributed to the problem that interrupts and faults were handled on user stacks. Specifically, when a timer interrupt occurred, and the handler invoked the scheduler to schedule another process, the kernel code suffered a page fault on its stack once it had switched to the address space of the new process. The EFB based handling of timer interrupts ensured that this process no longer happens on a user stack but on the expected kernel stack which is in kernel space and thus accessible from the address space of the new process.

Furthermore, unexpected behvaiour by the applications was observed in Valkey and memcached in the form of invalid responses to client requests and unexpected connection closures. These issues were likely caused by corruption of the application stack's red zone by interrupts, which is also resolved by the EFB based handling of interrupts.

More faults were encountered especially in the configurations that bracketed privilege. These faults typically surfaced as general protection faults due to a fault handler trying to access per-cpu data structures but having an inconsistent value in `gs`. The problem here was that there were several code locations in the transition code between user and kernel mode where the CPU operated in kernel mode (i.e., with `CS = 0x10`) but the `gs` register did not yet point to the kernel's per-cpu data structure. The same occurred when transitioning back to user mode, after resetting `gs` and before re-entering user mode. As described in section 2.3, this issue was resolved by minimizing the amount of code that runs in this inconsistent state, disabling interrupts

and ensuring that no page faults can occur in this state by aligning the transition code to a page boundary.

Another issue that was encountered specifically in the constant stack configurations of lebench was that when performing `read` or `write` system calls to/from a block device, the scheduler would be invoked on the user stack because the process would block on I/O. This would cause the same behavior as described above when rescheduling on a user stack. A mitigation for this issue is to remove competing processes from the run-queue of the core on which the application is running and make sure that the application does not migrate to another core. Then, the scheduler will not attempt a context switch to another process.

Finally, dispite the adaptions made to the runtime, a fault persists in memcached when it tries to extend the heap. This issue will be investigated in future work.

3.3 Performance figures

We close this section with a presentation of performance figures obtained from the different applications and configurations in this case study.

3.3.1 Lebench

read and write

The figures 3·1 and 3·2 show the latency of the `read` and `write` system calls, respectively, for different configurations and data sizes. The results show that certain configurations of the Kcut optimizations can reduce the latency of these system calls. For `read` this is clearer than for `write`, where the results contain deviations that require further investigation. This confirms the expectation that there is a decision

space for the optimizations, where a trade-off between performance and stability can be made.

For the `read` system call, the Kcut-BPBS and BPCS configurations show a increase in latency compared to the baseline across all data sizes. This is expected, as the bracketing of privilege introduces the overhead of one system call and one ring transition back to user mode. The Kcut-CPBS and CPCS configurations, on the other hand, show a reduction in average latency compared to the baseline. For smaller data sizes, this reduction is over 30%, while for larger data sizes, the reduction is closer to 20%.

For the `write` system call, there is a less significant difference in latency between the different configurations. The standard deviation of the latencies of Kcut-CPCS and the baseline are much higher and overlap more strongly. As opposed to the `read` there is a clear difference between the baseline and the base-IST configuration, which requires further investigation. Only looking at base-IST and the four Kcut configurations for data sizes up to 4KB, the a similar trend to the `read` system call can be observed, with the Kcut-CPBS and CPCS configurations showing a reduction in latency and the Kcut-BPBS and BPCS configurations showing an increase over base-IST.

recv and send

The figures 3·3 and 3·4 show the latency of the `recv` and `send` system calls, respectively, for different configurations and data sizes. The results show similar trends to the `read` and `write` system calls, with the Kcut-CPCS configuration showing the biggest reduction in latency compared to the baseline. However, there is a strong overlap in the standard deviation of the latencies of the Kcut-CPCS and the baseline,

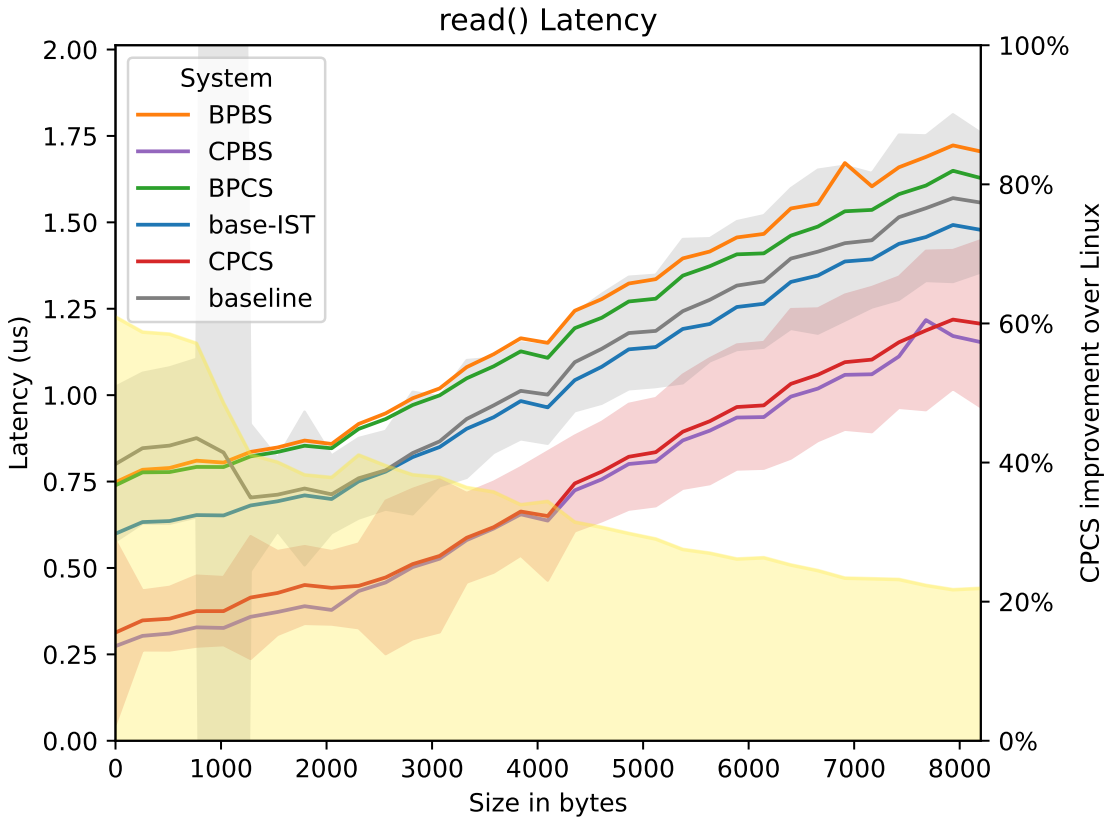


Figure 3·1: Lebench read latency for different configurations and speedup of Kcut-CPCS over the baseline. The standard deviation of the latencies of the baseline and Kcut-CPCS configurations is shown as shaded areas around the respective lines. The yellow shaded area shows the improvement of Kcut-CPCS over the baseline.

which requires further investigation.

For the `recv` system call, the Kcut-CPCS and CPBS behave similarly, with Kcut-CPCS achieving a reduction in latency of between 10% and 20% compared to the baseline. BPCS and BPBS meanwhile show an increase in latency compared to the baseline.

The `send` system call shows similar trends to the `recv` system call, with the Kcut-CPCS configuration showing the biggest reduction in latency. However, the baseline and base-IST configurations show strong fluctuations for all data sizes, which needs

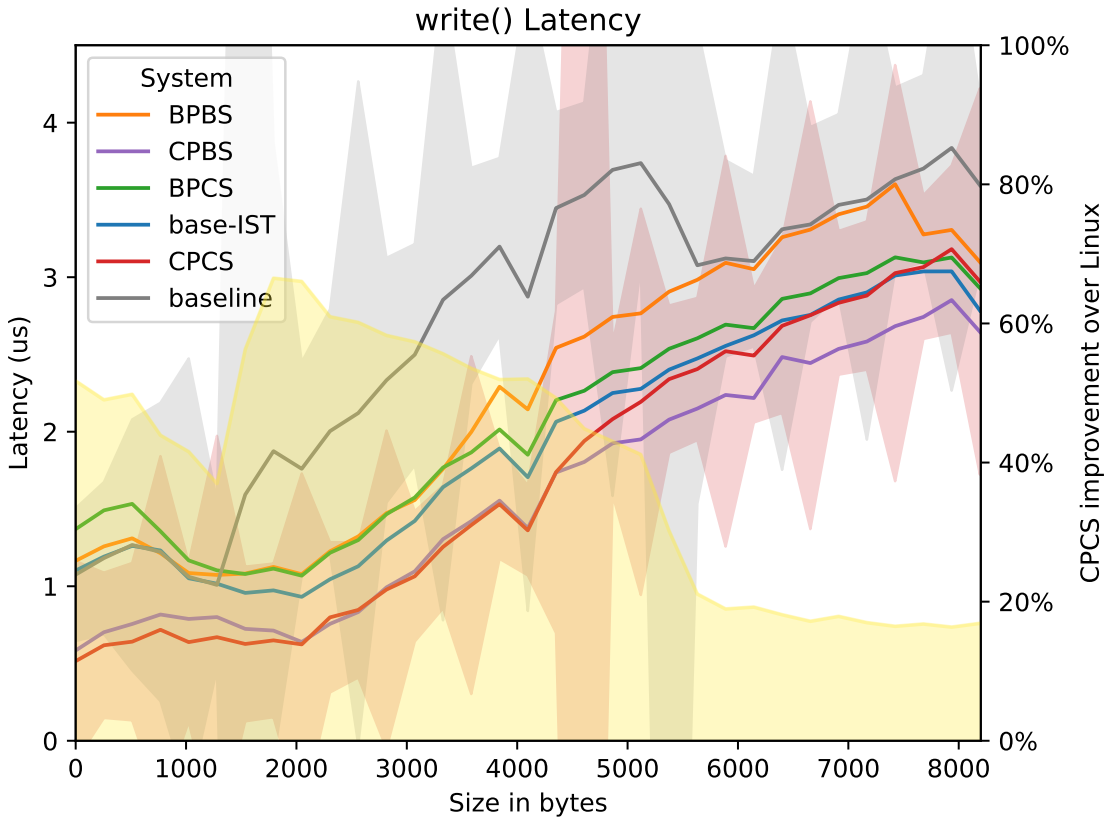


Figure 3·2: Lebench write latency for different configurations and speedup of Kcut-CPCS over the baseline. The standard deviation of the latencies of the baseline and Kcut-CPCS configurations is shown as shaded areas around the respective lines. The yellow shaded area shows the improvement of Kcut-CPCS over the baseline.

to be investigated further. Again, the Kcut-CPCS and CPBS configurations behave very similarly as well as the Kcut-BPCS and BPBS configurations. Kcut-CPCS shows a reduction in latency of between 10% and 20% compared to the baseline.

3.3.2 Valkey

The results obtained from the evaluation of Valkey show that the Kcut optimizations can be applied to real-world applications. Across all configurations, the application ran stable and showed correct behavior, as verified by the memtier benchmark.

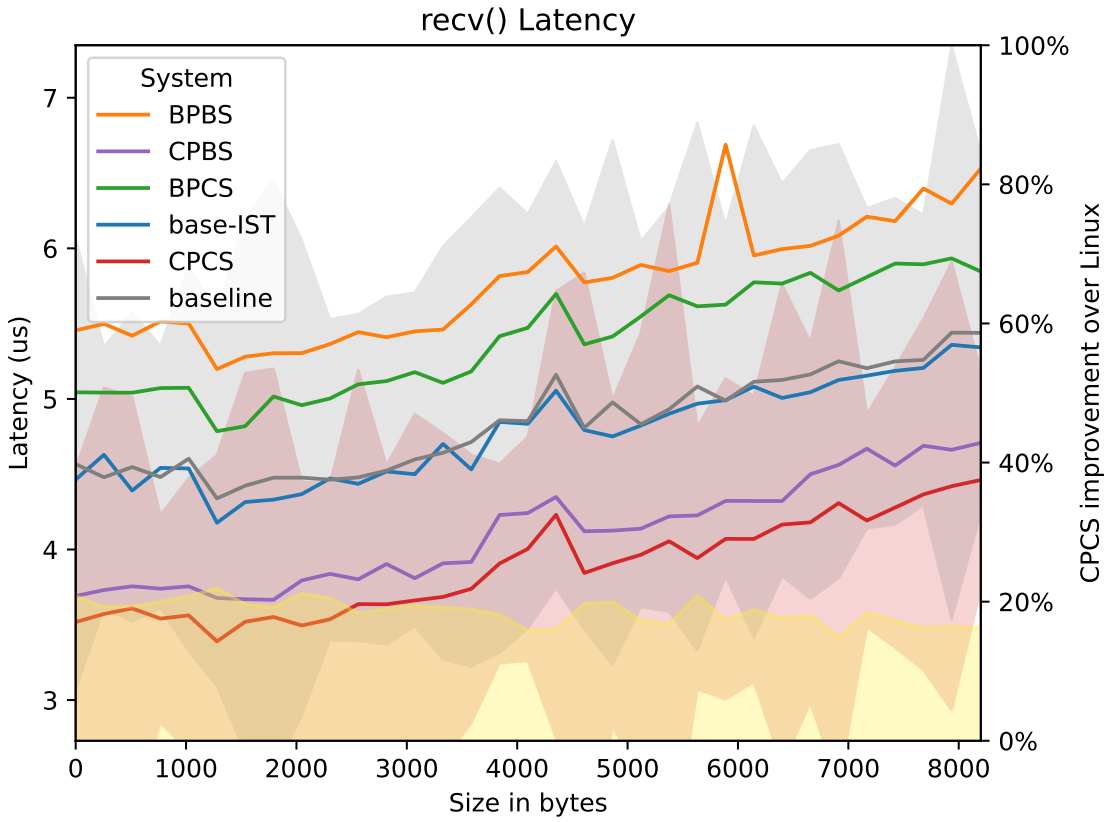


Figure 3·3: Lebench recv latency for different configurations and speedup of Kcut-CPCS over the baseline. The standard deviation of the latencies of the baseline and Kcut-CPCS configurations is shown as shaded areas around the respective lines. The yellow shaded area shows the improvement of Kcut-CPCS over the baseline.

The latency results for Valkey suffer from high variance within each configuration across multiple runs. The results are presented in Figure 3·5. For brevity, the results show only the baseline, Kcut-BPBS, and Kcut-CPCS configurations. It is notable that mean and 99%-tail latency of both the Kcut-BPBS and Kcut-CPCS configurations are higher than the baseline, which is unexpected based on previous research. Furthermore, it is notable that the Cumulative Distribution Function (CDF) of the Kcut configurations has a differnt shape than the baseline, with several inflection points.

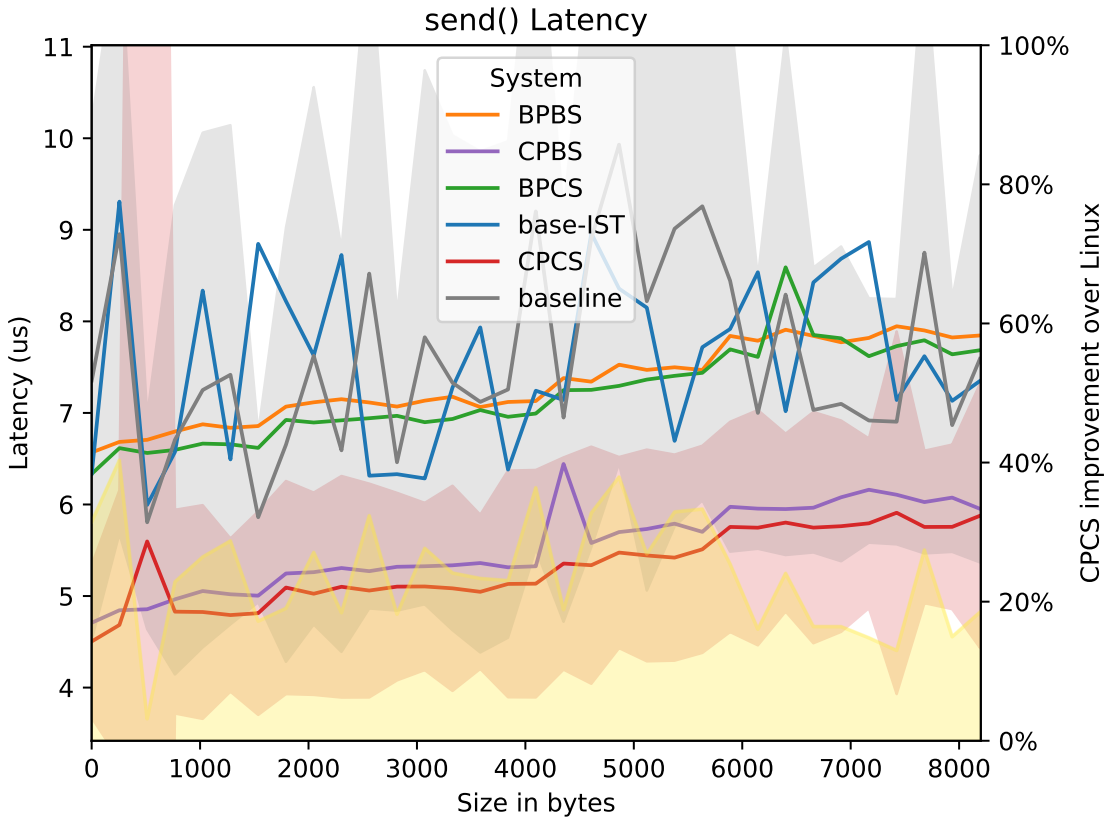


Figure 3·4: Lebench send latency for different configurations and speedup of Kcut-CPCS over the baseline. The standard deviation of the latencies of the baseline and Kcut-CPCS configurations is shown as shaded areas around the respective lines. The yellow shaded area shows the improvement of Kcut-CPCS over the baseline.

3.3.3 memcached

The results obtained from the evaluation of memcached reveal that it is possible to apply the Kcut optimizations to a multi-threaded application. The application ran stable in the Kcut-BPBS and Kcut-BPCS configurations and produced correct results, as verified by the memtier benchmark. However, the Kcut-CPCS and Kcut-CPBS configurations did not run as expected. The benchmark client reported several unexpectedly closed connections, and the server log reported unexpected page faults. The latency results for the baseline, Kcut-BPBS, and Kcut-BPCS configurations are

shown in figure 3·6. The results are in line with the results obtained from the microbenchmark lebench, with the Kcut-BPBS and Kcut-BPCS configurations showing an increase in latency compared to the baseline.

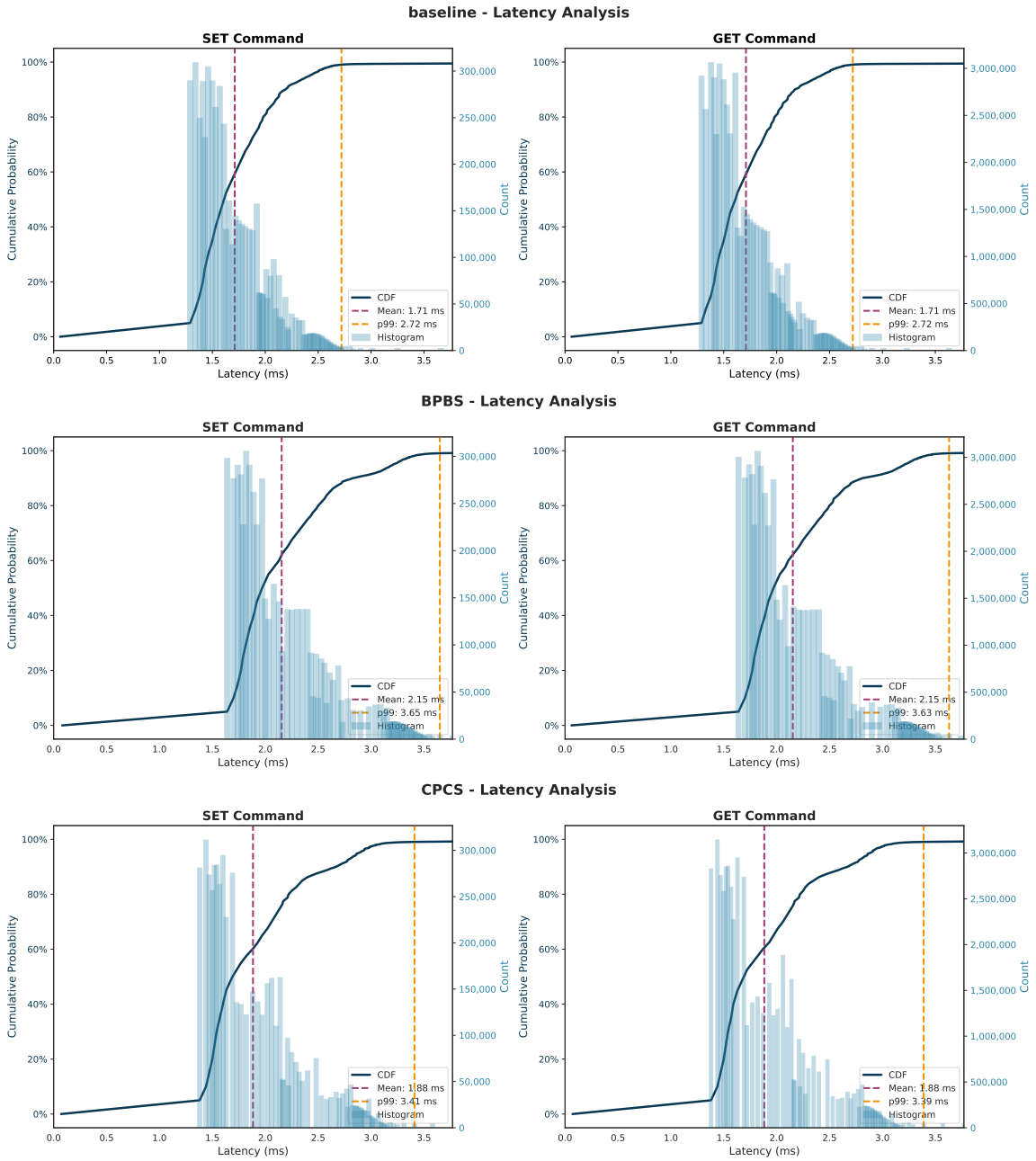


Figure 3·5: Valkey latency for baseline, Kcut-BPBS, and Kcut-CPCS configurations.

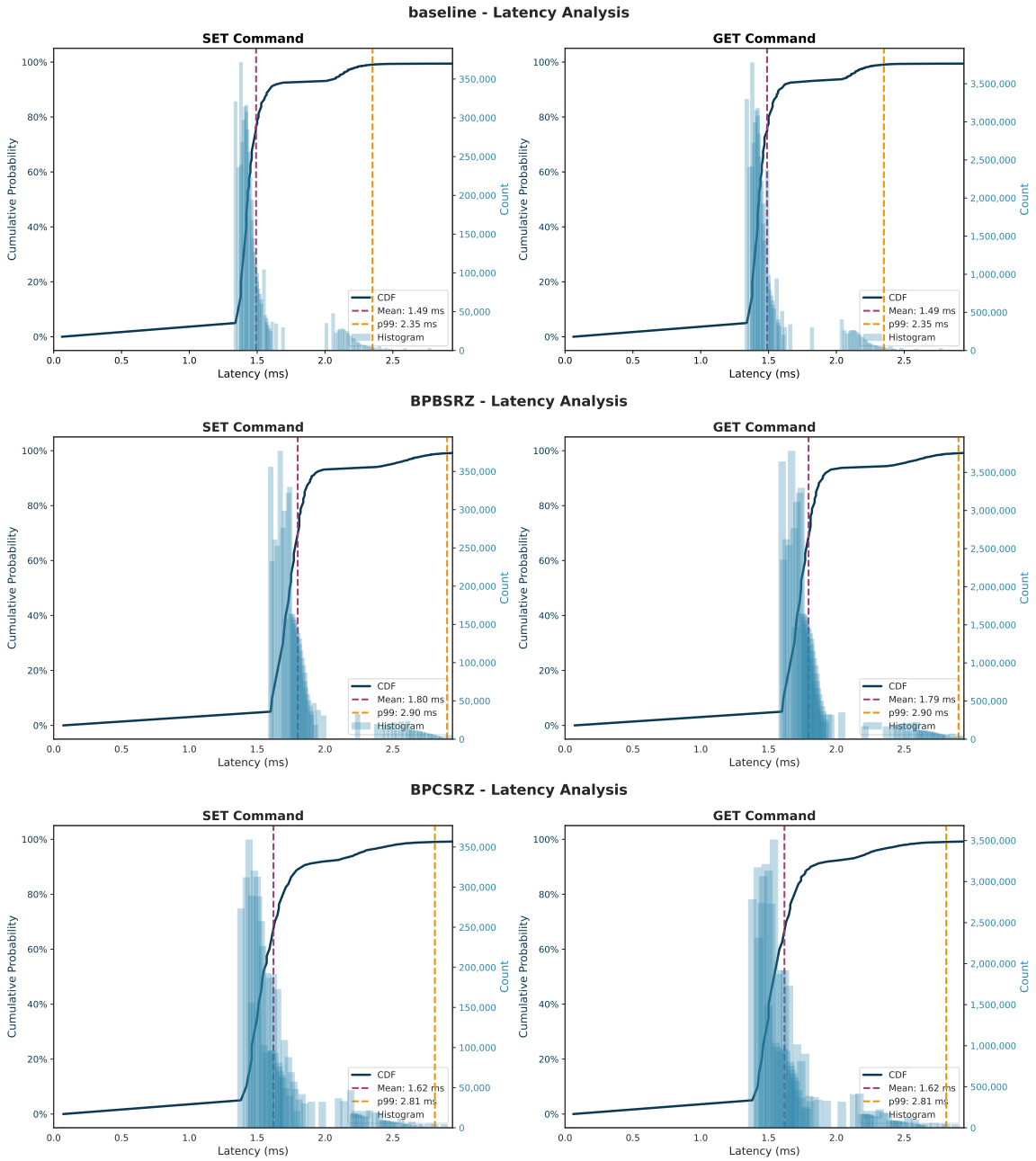


Figure 3·6: Memcached latency for baseline, Kcut-BPBS, and Kcut-BPCS configurations.

Chapter 4

Future Work

The work presented in this thesis offers multiple avenues for future work. The results presented in the previous chapter suggest that Kcut applications can achieve high I/O performance with high usability. However, further work is needed to fully evaluate the potential of dynamic privilege. It has been shown that there is a design space for Kcut applications consisting of the dimensions of depth, conformance, and privilege switch frequency. Exploring this design space and understanding the trade-offs between these dimensions is an interesting avenue for future work.

Furthermore, there are multiple opportunities for runtime optimization with respect to performance and security improvements.

4.1 Design Space Exploration

With regard to the applications considered in this thesis, the design space of depth, conformance, and privilege switch frequency is still largely unexplored.

Especially the dimension of depth offers a lot of potential for future work. For example, it would be interesting to let applications directly interact with NIC drivers to further optimize I/O performance. Similarly, storage applications could directly interact with Solid State Drive (SSD) drivers to optimize storage performance.

4.2 Comparative Evaluation

Given the plentitude of library OSs and kernel-bypass frameworks, it would be interesting to perform a comparative evaluation against the Kcut runtime using a set of applications that are implemented for all systems.

This would be especially interesting after performing the design space exploration to validate that the design space described above is sufficient or should be extended. For that, gathering large amounts of profiling data would be necessary to gain a deeper understanding of the performance characteristics of the different systems and applications.

4.3 Runtime Optimization

Optimizing the dynamic privilege runtime is another interesting avenue for future work.

The results presented in the previous chapter suggest that the overhead of privilege switches is a significant factor in the performance of Kcut applications. Therefore, optimizing the privilege switch mechanism to reduce this overhead could lead to significant performance improvements. This could be achieved by dedicating an interrupt vector to the privilege switch mechanism. Applications could then trigger a privilege switch using the `int` instruction. Depending on the implementation of the interrupt handler, this could significantly shorten the code path for a switch from user privilege to kernel privilege.

4.4 Hardware Support for Dynamic Privilege

Finally, it would be interesting to explore hardware support for dynamic privilege. Having a hardware mechanism that switches the privilege level without the need for

a context switch could further reduce the overhead of privilege switches. This would allow Kcut applications to increase their level of "conformity" without sacrificing performance.

Chapter 5

Conclusions

5.1 Summary of the thesis

This thesis explored the design and implementation of a novel approach to building high-performance applications that leverage the Linux kernel as a reusable library. Building on prior research in Dynamic Privilege and Unikernel Linux, this work introduced the Kcut runtime and toolchain, which enables developers to construct hybrid applications that combine standard user-space code with direct access to kernel code and data. The goal was to make the benefits of kernel short-cutting accessible to application developers in a way that is stable and familiar, without requiring a complete restructuring of the application build environment.

The thesis made three primary contributions. First, it introduced libKern.so, a novel mechanism that exposes the Linux kernel's symbol table to user-space linkers and loaders. This library, implemented as a kernel component, provides both static (`libkallsyms.a`) and dynamic (`libkallsyms.so`) linking capabilities to kernel symbols through virtual files in the /proc directory. This approach overcomes the limitations of previous implementations that relied on `System.map`, including incompatibility with KASLR, inability to resolve symbols from loadable kernel modules, and the need for custom symbol resolution code. By enabling standard linker toolchains to resolve kernel symbols, `libkallsyms` significantly simplifies the development of privileged applications.

Second, the thesis presented the HBB application model, which separates Kcut ap-

plications into two components: a user-space executable and a kernel-space extension implemented as a kernel module. This dual-component architecture resolves fundamental conflicts that arise when attempting to link user-space and kernel-space code using a single toolchain, including memory model conflicts arising from the kernel's assumption of 32-bit addressability and symbol conflicts between kernel and user-space libraries. The Kcut toolchain was developed to support this model, introducing new file extensions (`.kc`, `.kh`, `.kS`) to distinguish kernel-space source code and integrating the Linux kernel's build system with traditional user-space build workflows. The toolchain generates the necessary linkage infrastructure, including `ifunc` definitions for lazy module loading and GOT/PLT structures to facilitate bidirectional function calls between application components.

Third, the thesis enhanced the Dynamic Privilege runtime to increase the stability of Kcut applications. The IST stack interposition mechanism was introduced to prevent stack corruption and access violations that could occur when handling interrupts on user-space stacks. Privilege transition mechanisms were enhanced to maintain consistency between hardware privilege levels and the state of the `gs` register, preventing fatal errors during interrupt handling. Additionally, the runtime was forward-ported from Linux kernel 5.14 to 6.16, ensuring compatibility with modern kernel versions.

The evaluation demonstrated that the contributions indeed improved the usability and stability of the Kcut runtime. Using a set of three applications (a microbenchmark, a singled threaded key-value store, and a multi-threaded key-value store), the evaluation showed that Kcut applications can be developed using standard development tools and workflows. The applications were tested for their stability which revealed that the contributions have significantly improved the stability of Kcut applications. Finally, the performance data collected in the evaluation suggests that

further evaluation might reveal stable performance improvements for Kcut applications compared to their traditional counterparts.

This was pointed out as a clear direction for future work. Other directions for future work include further exploration of the design space of Kcut applications, comparative evaluations against existing library operating systems and kernel-bypass frameworks and further runtime optimizations and hardware support to reduce the overhead of privilege switches.

References

contributors, M. redis/memtier_benchmark: Nosql redis and memcache traffic generation and benchmarking tool. Accessed: 2026-04-09.

contributors, V. Valkey documentation. Accessed: 2026-04-09.

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

Foundation, L. Dpdk – the open source data plane development kit accelerating network performance. Accessed: 2025-10-13.

kernel development community, T. Extensible scheduler class. Accessed: 2026-04-13.

Madhavapeddy, A. and Scott, D. J. (2014). Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69.

NeXT Computer, Inc. (1993). *NeXTSTEP Developer's Reference, Volume 2*. NeXT Computer, Inc., Redwood City, California.

Raza, A., Unger, T., Boyd, M., Munson, E. B., Sohal, P., Drepper, U., Jones, R., De Oliveira, D. B., Woodman, L., Mancuso, R., et al. (2023). Unikernel linux (ukl). In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 590–605.

Ren, X., 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*, pages 554–569.

Thompson, T. (1989). The apollo domain series 10000. *Byte Magazine*, 14(1).

Unger, T. (2024). *Dynamic Privilege*. PhD thesis, Boston University. https://hdl.handle.net/2144/51234.

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36.

Zussman, T., Zarkadas, I., Carin, J., Cheng, A., Franke, H., Pfefferle, J., and Cidon, A. (2025). cache_ext: Customizing the page cache with ebpf. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 462–478.