Improving Operating System Security through Enhanced Isolation: Intra-Unikernel Isolation and Storage Server for a Multiserver-Library Operating System

Mincheol Sung

Preliminary Exam

Doctor of Philosophy

in

Computer Engineering

Binoy Ravindran, Chair

Matthew Hicks

Nikolaev Ruslan

Haining Wang

Haibo Zeng

May 11, 2021

Blacksburg, Virginia

Keywords: Operating Systems, Unikernels, Multiserver OS, Security, Virtualization

Improving Operating System Security through Enhanced Isolation: Intra-Unikernel Isolation and Storage Server for a Multiserver-Library Operating System

Mincheol Sung

(ABSTRACT)

In the computer system stack, security of the operating system (OS) is arguably one of the most important since it is a privileged and trusted entity that applications rely on. Modern computer systems have increasingly large OS code bases, and consequently expose a large attack surface. Isolation is a fundamental OS design principle, which can improve OS security, among others. The principle, first employed in microkernel OSs, isolates OS components in separate address spaces: a minimal set (e.g., process/memory management, scheduler, IPC) is executed in the CPU privileged layer and all others (e.g., device drivers, networking, storage) are executed in the unprivileged userspace layer. This enables localization of security exploits, improving security. The principle was subsequently studied in numerous other OS models including microkernel/multiserver OSs, exokernel/library OSs, kernel-bypass techniques, and most recently in unikernel OSs.

In this dissertation, we present two techniques to enhance isolation in unikernel and multiserver-library OSs. First, we present a technique for memory isolation *within* a unikernel OS. In the unikernel OS model, an application is statically compiled together with the minimal necessary OS components, programmed as a library OS, and executes in a single address space. The resulting reduced code base and attack surface paradoxically also reduce security: security vulnerabilities in unsafe kernel code or user code can be exploited to compromise kernel code as they are not separated. Our technique prevents such exploits through their

isolation, while retaining unikernel's single address space model. The technique leverages recent commodity hardware primitive (i.e., Intel's Memory Protection Keys) which enables per-thread permission control over groups of virtual memory pages. We leverage language macro features to provide unikernel library OS developers with a convenient way to annotate code for isolation. Our implementation in the RustyHermit unikernel and evaluations reveal that the technique achieves isolation with only 0.6% slowdown.

Second, we present a storage server in LibrettOS. LibrettOS combines the multiserver and library OS models in the same OS design, enabling multiserver OS's high isolation and library OS's high performance in the same OS, and the ability to dynamically switch between the two models at run-time for more effective resource utilization. LibrettOS lacks a storage server, an essential system server in multiserver OSs. We present the design and implementation of a storage server for LibrettOS that transfers inbound and outbound block I/O between applications and the storage device, while isolating applications and the storage device, localizing security vulnerabilities. We build the storage server using the rumprun unikernel, which allows us to leverage NetBSD's NVMe driver with little engineering effort. We leverage lock-free ring buffers and design an efficient IPC mechanism. Our evaluations reveal that the storage server achieves comparable or superior performance compared with rumprun and Linux.

Contents

List of Figures

List of Tables

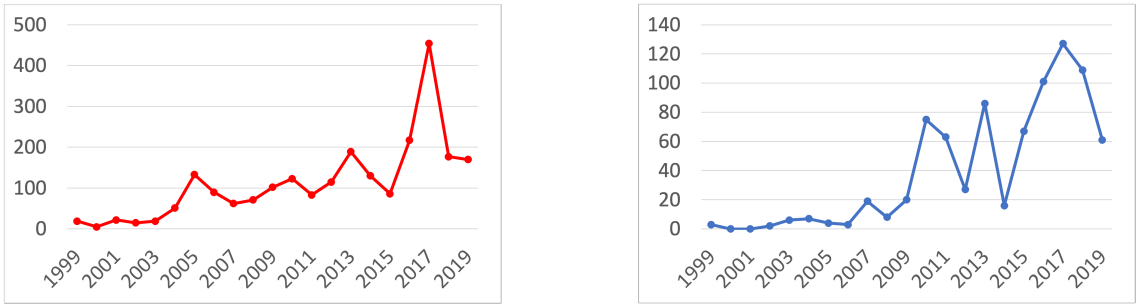
Chapter 1

Introduction

Security exploits of computer systems are now ubiquitous. In the computer system stack, the security of the operating system (OS) is arguably one of the most important since it is a privileged and trusted entity that applications rely on. Even though applications themselves may take measures to defend against security threats, a vulnerable OS can easily jeopardize those measures.

Traditional OSs execute all OS components in a single privileged layer for better performance. This results in a monolithic design wherein OS subsystems such as that for process management, memory management, file systems, storage, and networking, among others, as well as all device drivers are given the same level of trust and thus execute with the same CPU privilege within a single (kernel) address space. The monolithic design is increasingly inadequate for modern OSs as their code size have become ever more larger. For example, Linux, one of the most widely used monolithic OSs, has about 28 million lines of code [61].[1] Broadly described, a large kernel code base inevitably exposes a large attack surface: any security vulnerability in a kernel component can be exploited to compromise the entire system. The number of common vulnerabilities and exposures (CVEs) for kernel subsystems and device drivers continues to surge across popular monothilic OSs (see Figure 1.1). Tolerating component failures with monolithic OS designs is also a challenge as a single component failure can fail the entire system.

[1] This is for Linux kernel version 5.5.



(a) Linux kernel [31]

(b) Windows kernel [26]

Figure 1.1: Number of CVEs of Linux and Windows kernels.

To improve OS security (and also reliability), a fundamental design principle studied in the OS literature is *isolation*. Microkernel OSs [66], which first explored this principle, isolated OS components in separate address spaces: minimal kernel components (e.g., process and memory management, scheduler, IPC) are executed in the privileged layer and all other components including device drivers are executed, together with applications, in the unprivileged user space layer. This enables localization of security exploits and also component failures. Multiserver OSs [30, 42, 45], which can be viewed as a type of microkernel design, implement specific kernel subsystems such as file systems, storage subsystems, networking, and device drivers as user space server processes. Applications communicate with the servers using IPC. Although these OS designs improve security and reliability through compartmentalization and isolation, they come at the price of performance. Crossing privilege layers through IPCs, system calls, and mode switches are expensive, and consequently, microkernels and multiserver OSs are not performance-competitive to monolithic OSs [40, 64].

Performance limitations due to the need to cross privilege layers can be overcome by implementing kernel components as libraries; applications are built by composing them together with the kernel libraries. This model, first proposed by Tom Anderson [6], was later implemented in the exokernel model [34], and named the "library OS." Kernel-bypass techniques as exemplified in networking libraries such as DPDK [108] and storage libraries such as

SPDK [101] are a form of library OS. By avoiding kernel code in performance-critical data paths, they reduce the overhead of the kernel code and also the need to switch privilege modes and thereby improve performance. Traditionally, kernel-bypass techniques have introduced their own custom APIs and no standardized APIs (like POSIX extensions) exist. Thus, applications must be significantly modified to use them, which usually requires massive engineering efforts.

Unikernels, a relatively new OS model [68], can be viewed as a specialized form of library OSs. A unikernel instance includes a single application that is statically compiled together with the necessary kernel components and executes in a single address space. Since traditional system calls are replaced with regular function calls [27, 81], mode switch overheads [100] are avoided, improving performance. Since each unikernel instance houses a single application, the resulting code base and the attack surface are significantly smaller than monolithic OSs, improving security. Unikernels are usually executed in a virtualized environment, atop a hypervisor, which provides strong isolation – usually leveraging hardware virtualization – between different unikernel instances, further improving security. Given these benefits, they have recently gained traction in a number of domains including cloud/edge computing [15, 54, 55, 70, 99], server applications [55, 68, 69, 99, 120], NFV [28, 68, 70, 71], IoT [28, 33], HPC [59], VM introspection and malware analysis [118], and regular desktop applications [85, 109].

1.1 Intra-Unikernel Isolation

Although the isolation between unikernels is generally recognized as strong due to virtualization, there is no isolation *within* a unikernel. This is due to the use of a single unprotected address space, as the model eliminates the traditional separation between kernel and user parts of the address space. Thus, the entire unikernel must be viewed as a single unit of trust,

reducing security: subversion of a kernel or application component will result in the subversion of the entire unikernel with serious consequences, such as arbitrary code execution, critical data leaks or tampering, among others.

We argue that the current level of isolation provided by unikernels is too coarse-grained for many scenarios. First, a single application may be composed of mutually untrusting components [8, 111], e.g., if they came from different sources with variable security coding standards. Second, although some library OSs are written in memory-safe languages [21, 60, 68, 115], they generally rely on untrusted components for low-level operations by using a traditional unsafe language [21, 68, 115] or by using `unsafe` code blocks in languages such as Rust. Some unikernel library OSs are written entirely in an unsafe language [49, 51, 59]. Third, in scenarios where mutually untrusting components belonging to the same application need to be isolated [8, 41, 111], a computing base that is trusted from the tenant's point of view – as in a cloud setting – has to be established to enforce that isolation. In the current state of the unikernel model, this trusted computing base (TCB) cannot be the guest OS kernel as it is not itself isolated from the application. This implies falling back to the hypervisor as the TCB, which is suboptimal from a performance standpoint.

Several isolation mechanisms have been studied in the past to isolate an application's untrusted components. These mechanisms operate at various levels of abstraction: a) using hardware-assisted virtualization [11, 53], b) running components in different processes or using different page tables [8, 58], and c) using ISA extensions that support a trusted execution environment such as Intel SGX [7, 16]. None of these mechanisms can easily be applied to unikernels without breaking the single address space model, which would not only negate unikernels' performance benefits but also introduce additional, non-negligible performance overheads in the form of mode switching costs [111].

In this dissertation, we address the aforementioned security issues of unikernels by provid-

ing intra-unikernel isolation while retaining their single address space feature. We leverage hardware primitives that are available in commodity processors to do this. Specifically, we use Intel's Memory Protection Keys (or MPK) [23] primitive, which provides per-thread permission control over groups of virtual memory pages in a single address space. MPK has negligible performance overheads [82, 111], which makes it a compelling candidate for use in unikernels.

We identify the different areas of a unikernel's address space, i.e., the kernel's safe/unsafe memory regions (static data section, stack, and heap), and the user memory regions (static data section, stack, and heap). These areas are isolated from each other by using MPK-based mechanisms to enforce per-thread permissions on each memory area. In designing the isolation mechanisms, our design principles are: (1) they should preserve a single address space; (2) they should isolate the various areas of the address space; and (3) they must have negligible performance overheads.

We demonstrate our mechanisms in RustyHermit [60], a unikernel written in Rust. We use easy-to-use code annotations for identifying intra-unikernel components that require isolation. Such annotations can be made by the unikernel library OS programmer. We design and implement two isolation policies. First, we isolate safe Rust kernel code from unsafe Rust kernel code to limit the possibilities for an attacker to exploit a vulnerability in the unsafe kernel code. Thus, trusted kernel components are protected from attacks that leverage vulnerabilities in untrusted ones. Second, we re-introduce kernel and user space separation in unikernels by isolating kernel code from user code. This protects kernel space from unauthorized access by subverted user code. Additionally, it enables implementation of isolation techniques that isolate application components from each other, which should be enforced by the kernel. Our isolation techniques have low overhead. In particular, they retain unikernels' low system call latency feature.

1.2 Storage Server in Multiserver-Library OS

No single OS model is ideal for all use cases. As previously discussed, monolithic OSs tradeoff isolation (and thereby potentially reduced security and reliability) for high performance. Microkernels and multiserver OSs tradeoff performance for high isolation. Library OSs and kernel-bypass techniques tradeoff programmability (usually POSIX) for high performance.

Multiple OS models coexisting in the same OS have a value proposition: each model's advantage can be exploited, allowing "best of several worlds". In addition, applications can switch between different OS models, at run-time, for more effective resource utilization. This is particularly significant when using limited resources – e.g., modern 10 GbE Ethernet NICs [47] often have limited SR-IOV interfaces [83]. Thus, at low I/O loads, a multiserver OS model for device access can be performant; when I/O load increases, however, the library OS model may become necessary for high performance and can be dynamically switched to.

Motivated by these considerations, we introduced LibrettOS [79], the first OS design that supports multiserver and library OS models in the same OS, and the ability to switch between the two models at run-time. LibrettOS's default mode is the multiserver OS mode. Applications that require strong isolation for better security and reliability can leverage LibrettOS's servers that implement OS subsystems. For selected applications that require high performance, LibrettOS's direct mode acts as a library OS and allows applications to directly access hardware resources. In addition, OS subsystem code, such as that of device drivers, network stacks, and file systems remain identical in the two modes, enabling applications to interface with them using the same set of APIs, in fact, POSIX APIs, enabling dynamic mode switching with significantly reduced development and maintenance costs.

LibrettOS is built using the rumprun unikernel [102], which is based on NetBSD 9.0 [74]'s code base, was introduced by Antti Kantee [48], and upstreamed into mainline NetBSD [73].

This allows us to reuse existent, hardened NetBSD device drivers with little engineering effort. In addition, LibrettOS supports standardized high-level APIs, e.g., POSIX, which allows a large ecosystem of POSIX/BSD-compatible applications to be ported without any modification.

LibrettOS [79] was demonstrated using a network server implementation, and used Xen [50] as the underlying hypervisor in HVM mode [116] for server isolation. The network server leverages NetBSD's 10GbE driver code along with a small glue code. Experimental evaluations [79] using applications including Nginx, NFS, memcached, Redis, and others demonstrated LibrettOS's superior performance over the original rumprun and NetBSD, especially in the direct mode. In some tests, LibrettOS also outperformed Linux, which is often better optimized for performance than NetBSD.

Along with the network server, the storage server, which LibrettOS lacked, is one of the basic and essential system servers in multiserver OSs. A storage server's basic functionality includes transferring inbound and outbound block I/O between applications and the storage device. In the multiserver OS model, such an isolation enhances security.

This dissertation presents the design and implementation of a storage server in LibrettOS. We design the storage server using the rumprun unikernel and by breaking-down functions of the block layer and by reusing the storage device driver (e.g., NVMe driver [75]) from the NetBSD kernel. The storage server adopts the paravirtualized driver [117]. The frontend driver is linked as a library in the application's address space so that the application can communicate with the storage server. On the storage server-side, the backend driver is linked as a library along with other device drivers such as the NVMe driver. The frontend and backend drivers communicate through an IPC. We also implement an efficient IPC between the storage server and applications by leveraging lock-free ring buffers [77].

1.3 Summary of Research Contributions

In summary, this dissertation presents two techniques for improving OS security through enhanced isolation:

1. *Intra-Unikernel Isolation with Intel MPK.* A single address space is unikernel OS's fundamental design principle for high performance and enhanced isolation. Paradoxically, this can reduce security: since unsafe kernel code and safe kernel code are not separated, and user code and kernel code are not separated, security vulnerabilities in unsafe kernel code or user code can be exploited to compromise kernel code. We present a technique to achieve intra-unikernel isolation that prevents such exploits by leveraging Intel's MPK hardware primitive. We leverage language macro features to provide unikernel library OS developers with a convenient way to annotate code for isolation. Our implementation in the RustyHermit unikernel and evaluations reveal that the technique achieves isolation with very low performance overhead: 0.6% slowdown on applications including memory/compute intensive benchmarks (NPB [94], PARSEC [13], Phoenix [91]) as well as micro-benchmarks.

2. *Design and Implementation of Storage Server in LibrettOS.* LibrettOS is an OS design that supports multiserver and library OS models in the same OS, allowing each model's advantage (i.e., high isolation, high performance) to be exploited, and the ability to dynamically switch between the two OS models at run-time for more effective resource utilization, using the same set of (POSIX) APIs. LibrettOS lacks a storage server, an essential system server in multiserver OSs. We present the design and implementation of a storage server in LibrettOS that enhances isolation between applications and the storage device, thereby allowing localization of security vulnerabilities. We build the storage server using the rumprun unikernel, which allows us to leverage NetBSD's

NVMe driver with little engineering effort. To minimize performance degradation due to IPC, we leverage lock-free ring buffers and design an efficient IPC mechanism.

Our evaluations using applications including the NFS server and Sysbench benchmark reveal that the storage server outperforms Linux in certain cases and achieves reasonable performance compared with rumprun and Linux in other cases.

1.4 Summary of Proposed Post-Preliminary Exam Work

We propose the following directions for post-preliminary examination work:

- In the multiserver OS model, better isolation can not only improve security, but can also help tolerate component failures, increasing reliability. LibrettOS's storage server currently cannot transparently tolerate server failures. While simple reboots are a satisfactory solution for transparently tolerating network server failures, and was demonstrated in [79], they are not acceptable in the storage context. This is because storage is internally stateful: I/O requests that are in-flight when server crashes must be carefully handled so that no errors manifest in applications when the server is rebooted. In contrast, when a network server is rebooted, TCP/IP automatically resets the network connection and retransmits packets. We therefore propose to enhance LibrettOS's current storage server with a transparent fault-recovery mechanism.

- Similar to the storage server, a file system server can enhance isolation and thereby security and reliability in the multiserver OS model. A file system server's basic functionality includes transferring inbound and outbound file I/O between applications and a file system. We propose to design and implement a file system server for LibrettOS. Similar to the storage server's proposed fault-recovery mechanism, we also propose a mechanism for the file system server that can transparently tolerate server failures.

- The dissertation's intra-unikernel isolation mechanism can be further enhanced to achieve isolation at finer granularities. A case in point is isolation of the user library. A vulnerability in a user-facing HTTP parsing module (e.g., NGINX's CVE2013-2028 [2]) can leak sensitive data of the cryptographic library (e.g., crpyto keys). Thus, isolating a unikernel's user library code from a user application code can prevent leakage of sensitive data against a malicious client. We propose to design such fine-grained intra-unikernel isolation mechanisms.

- Besides network, storage, and file system servers, device servers such as that for USB, sound, and console can be designed for LibrettOS, further enhancing isolation. Modern hardware is equipped with a controller that has sufficient computational power to run a device server. The device servers can run on the hardware's controller such that they provide strong isolation of the device drivers. We propose to design and implement such device servers for LibrettOS.

1.5 Dissertation Organization

The rest of the dissertation is structured as follows:

Chapter 2 presents the necessary background of the dissertation. Chapter 3 reviews past and related works.

Chapters 4 and 5 present the design and implementation of the intra-unikernel isolation technique, respectively. Chapters 6 and 7 evaluate the security and performance of the intra-unikernel isolation technique, respectively.

Chapters 8 and 9 present the design and implementation of LibrettOS's storage server, respectively. Chapter 10 evaluates the storage server.

Finally, Chapter 11 concludes and proposes post-preliminary exam work.

Chapter 2

Background

In this chapter, we provide background information required to understand the rest of the dissertation. We first describe some background information about Unikernel and RustyHermit [60] that we work with in this dissertation, Rust [90] that is the programming language used to write RustyHermit, and the Intel MPK [23] technology that we use to provide isolation. We also provide background knowledge for the storage server. This includes Xen hypervisor, rumprun unikernel, and LibrettOS [79].

In this chapter, Section 2.1 discusses about unikernel. Section 2.2 introduces RustyHermit-core. Section 2.3 presents Rust programming language. Section 2.4 explains Intel Memory Protection Keys. Section 2.5 describes rumprun unikernel. Section 2.6 covers Xen hypervisor. Lastly, Section 2.7 explores LibrettOS.

2.1 Unikernel

A unikernel [68] consists of a single application compiled and statically linked with a minimal kernel LibOS. Unikernels are single purpose, i.e. one instance corresponds to one guest VM running a single application on top of a hypervisor. A unikernel instance also presents a single and unprotected address space shared between the kernel and the application. All the code executes with the highest privilege level (for example, ring 0 in x86-64) and thus there is no memory protection between kernel and user code/data in that address space.

Such a model brings significant benefits in several domains [81], in particular in terms of performance [51, 59, 81]: due to the elimination of kernel/user separation, system calls can be replaced with regular function calls. This significantly reduces system call latency, as there is no longer a costly world switch between privilege levels [59]; expensive operations such as page table switching [81] are eliminated. As a result, unikernels have been shown to outperform traditional OSs in system-intensive workloads [27].

However, the lack of isolation within a unikernel (intra-unikernel isolation) raises serious security concerns. Even if it executes a unique application, viewing a unikernel instance as a single and atomic unit of trust is too coarse-grained in current scenarios: a vulnerability in a relatively untrusted/vulnerable application component automatically leads to the attacker taking over the entire system. In a unikernel, this concern also includes kernel components, as there is no isolation between kernel and user space. We divide intra-unikernel isolation issues into two categories: (1) the lack of isolation between kernel and user space and (2) the lack of isolation between trusted and untrusted kernel components in memory-safe unikernels.

Lack of Isolation between Kernel and User Space. Modern applications are made of components (such as libraries) having variable degrees of trustworthiness/potential for vulnerabilities, manipulating data with various levels of sensitivity [8]. Without isolation between these, taking over a vulnerable component gives the attacker control over the entire application, including the sensitive data belonging to other components. Consider, for example, a formally verified cryptographic library [121] and a user-facing HTTP parsing module. The former is unlikely to contain vulnerabilities, but the sensitive data it manipulates (crypto keys) could be leaked through a vulnerability in the latter (such as CVE-2013-2028 in Nginx) when they run in the same application. Another example is an image manipulation library overwriting sensitive function pointers in the Global Offset Table [8]. To provide more security in these scenarios, intra-application solutions have been proposed [8, 41, 111]. They rely

on a trusted entity to enforce an isolation policy. Due to the lack of user/kernel separation in unikernels, that entity cannot be the guest kernel as the application code can freely access kernel memory. Although the hypervisor could play that role, it would be sub-optimal from a performance point of view (more VMEXITs). It would also lead to an increase in the trusted computing base (hypervisor), which is a security concern. In conclusion, to support isolation of components within applications, *it is necessary to bring back user/kernel separation in unikernels.*

Lack of Isolation between Trusted and Untrusted Kernel Components. Several unikernels' OS layers are written in memory-safe languages [21, 60, 68, 115]. This offers strong security guarantees compared to unikernels written in unsafe languages such C/C++ [49, 51, 59, 81]. However, even memory-safe unikernels rely on untrusted components to realize the low-level operations that are unavoidable in an OS context: the use of inline assembly and the need to dereference raw pointers. This is realized either with an unsafe language for those components [21, 68, 115] or with the use of unsafe code blocks [60] in a language such as Rust. Once again, without intra-unikernel isolation, a vulnerability in an unsafe kernel component leads to the subversion of the entire system, in effect negating the benefits of using a memory-safe language.

In the rest of this dissertation, we focus on RustyHermit [60], a unikernel which is written in Rust, although our design could easily be adapted to other unikernels that use C for low-level operations. One of the main reasons we chose RustyHermit for our implementation is the fact that, contrary to other memory-safe unikernels, it does not restrict the application code to the same language as the LibOS (such as OCaml for MirageOS, Erlang for LING, and Haskell for HaLVM), which is a significant compatibility advantage.

Listing 1 shows an unsafe code snippet extracted from RustyHermit's source code. These

```
impl<T> PerCoreVariableMethods<T> {
    #[inline]
    default unsafe fn get(&self) -> T {
        let value: T;
        asm!("movq %gs:($1), $0"
            : "=r"(value) : "r"(self.offset())
            :: "volatile");
        return value;
    }
    #[inline]
    default unsafe fn set(&self, value: T) {
        asm!("movq $0, %gs:($1)"
            :: "r"(value), "r"(self.offset())
            :: "volatile");
    }
}
```

Listing 1: Per-core variable get/set methods.

functions manage per-core variables using the GS x86-64 segment register, plus a relative offset depending on the variable. Examples of per-core variables are the CPUID, scheduling data structures, and task state segments. Practical addressing relative to the GS register can only be done using inline assembly, i.e. it should be placed within an unsafe code block. If we assume that, through a bug, the attacker has control over the `self` parameter, then the `set` function can be used to perform arbitrary memory writes (note that `self.offset()` returns a value deterministically computed from the value of `self`). Similarly, if we additionally assume that the attacker can exploit a bug to return the value of the `get` function, then it becomes an arbitrary memory read.

To conclude, in addition to kernel/user separation, there is also the need to bring isolation between safe and unsafe kernel components into memory-safe unikernels. Furthermore, neither type of isolation should come at the cost of degraded performance, nor should they negate the performance benefits of unikernels such as fast system calls, fast context switches,

and the like.

2.2 RustyHermit

The unikernel RustyHermit is completely written in Rust and does not depend on any C code. One of Rust's major advantages for kernel developers is that it splits the runtime into an operating-system-independent library and an operating-system-dependent library. By implementing Rust's global memory allocator, the *alloc* library, multiple data structures become available and usable in kernel space. These include smart pointers as well as basic data structures like linked lists, binary heaps, ring buffers, and maps. Only a target specification file that specifies processor type, pointer width, etc. is required to compile these libraries. Consequently, kernel developers can reuse existing, well-tested code from the Rust community, which simplifies development and increases the robustness of the kernel.

Additionally, RustyHermit is a full 64-bit kernel, supporting x86-64 processors, SIMD instructions like AVX, thread-local storage, and symmetric multiprocessing. RustyHermit is completely integrated into the Rust compiler infrastructure. One part of the Rust infrastructure is Cargo, which is Rust's package manager and coordinates the build process of Rust binaries. The main difference from the typical C/C++ build process is that the package manager does not install binaries, headers, static or shared libraries. It instead downloads the source code, compiles it with the same compiler flags, and links it directly to the executable. The Rust community calls such packages *crates*. By fully integrating RustyHermit into the Rust toolchain, cargo can be used to define the dependencies for the application. In principle, every published crate in a repository (e.g., crate.io) can be used to build executables based on the library operating system. The only requirement is that the crate must not directly call the host OS and bypass Rust's standard runtime.

Besides the support for pure Rust binaries, it is also possible to develop C/C++ applications

on top of the Rust kernel. For this purpose, the C library *newlib* [76] is used to create the interface between C/C++ applications and the kernel.

In addition, RustyHermit comes with the lightweight hypervisor *uhyve*, which is also completely written in Rust and uses KVM to accelerate the virtualization. RustyHermit can delegate operating system services like file system access to the host system by hypercalls. The technique is outlined in [81]. RustyHermit is composed of about 20k LoC, including 650 lines of unsafe code [60]. We implemented our intra-unikernel isolation technique in a RustyHermit unikernel running on *uhyve* hypervisor.

2.3 Rust

Rust is attracting attention as a system programming language because of the memory safety guarantees provided by its compiler. Furthermore, the absence of a garbage collector allows Rust to avoid much runtime overheads[29]. Instead of collecting unused memory in the runtime, Rust is designed to rely on comprehensive safety checking at compilation time; there are also runtime safety checks when the compile-time checks are not sufficient [60]. The concept of *ownership* ensures that all objects are safely handled with minimal runtime overhead. Thanks to Rust's memory safety and high performance, operating systems like RustyHermit [60], Theseus [14], TockOS [63] and Redox [32] were written in Rust.

Rust prohibits dereferencing raw pointers for memory safety. It is, however, inevitable for the kernel to access unchecked raw pointers, such as when accessing the page table. In some cases, the kernel has to call assembly, such as when executing start-up code directly. To support those cases, Rust also provides an unsafe code region that is not checked by the Rust compiler or runtime. As Rust's memory safety is not guaranteed in the unsafe block by the compiler, developers have to write vulnerability-free codes by themselves.

2.4 Intel Memory Protection Keys (MPK)

Intel Memory Protection Keys is a new hardware feature providing per-thread permission control over groups of pages without requiring modification of page tables at a small performance cost. Four previously unused bits of each page table entry (the 62nd to the 59th on x86-64) are exploited by MPK [23, 82]. Since MPK exploits four bits of the page table entry, it supports up to 15 protection keys (we opted to reserve key 0).

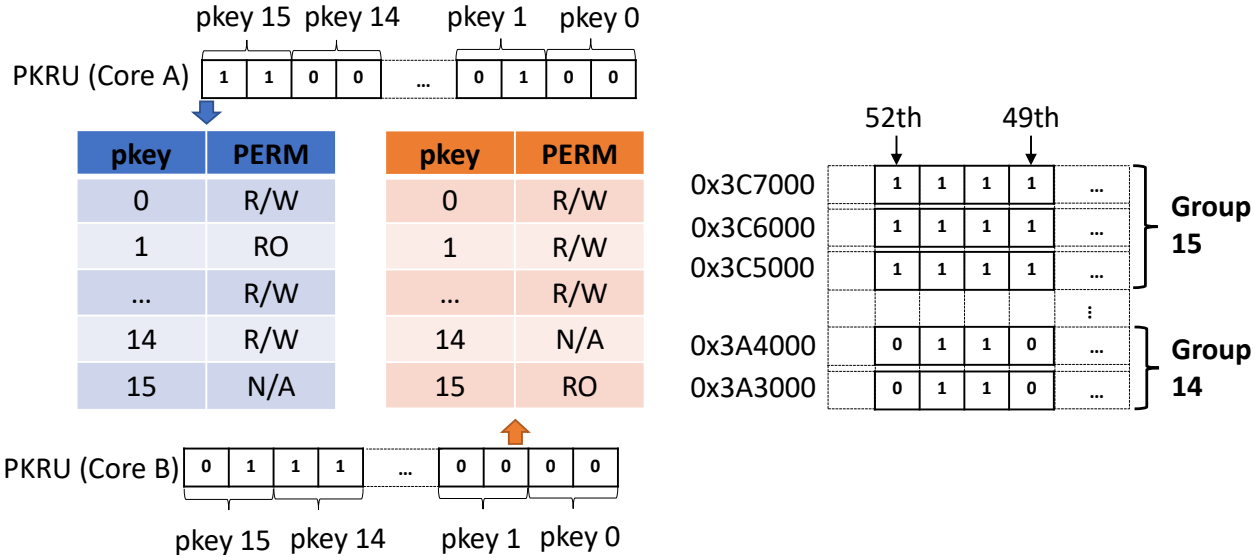


Figure 2.1: Illustration of Intel's Memory Protection Key (MPK) feature.

MPK controls per-thread permission on groups of pages with the notation (WD, AD), where WD is *Write Disable* and AD is *Access Disable*. The possible states are are read/write (0,0), read-only (1,0), or no-access (x,1). Each core has a PKRU register (32 bits) containing a permission value. The value of the PKRU register defines the permission of the thread currently running on that core for each group of pages containing a protection key in their page table entries. Figure 2.1 illustrates MPK's operation. A thread running on a core A has the *no-access* permission on the pages of group 15 and *read-write* on those of group 14. On the other hand, a thread running on core B *can not access* the pages of group 14 and

can *only read* the pages of group 15.

Unlike page-table-level permission, MPK provides thread-local memory permission. Furthermore, the cost of switching the PKRU value is quasi-negligible [111]. We believe MPK is most suitable for providing isolation within a unikernel without harming the principle of unikernels.

2.5 Rumprun

Rumprun is a unikernelized implementation of rump kernel [49] whose code is based on NetBSD's kernel code. Even though NetBSD is a well-known monolithic OS, the rump kernel is designed to be modular such that NetBSD's device drivers can be factored out. Additional rumprun layer enables the rump kernel to run as a unikernel on Xen hypervisor, KVM, or a bare-metal machine. The major reason that we opted to use rumprun unikernel is that it can leverage a wide range of NetBSD's device drivers maintained by the NetBSD community. Our storage server is implemented on a rumprun unikernel running on Xen hypervisor. As a result, it relies on Xen hypervisor's subsystems such as grant references and event channels. In addition, rumprun can run as either of PV (Paravirtualization) domain or HVM (Hardware Virtualization Mode) domain on Xen. We build our storage server on rumprun HVM mode.

2.6 Xen Hypervisor

Xen hypervisor [50] is a type-1 hypervisor first introduced and developed by the Computer Laboratory at the University of Cambridge. Xen hypervisor firstly brought paravirtualization which requires guest operating systems modified to run on Xen but achieves better performance, unlike the full virtualization. Domain means a guest machine in Xen term. Xen hypervisor has a privileged domain called Dom0 which is running Linux, NetBSD, or

other well-known operating systems. The responsibility of Dom0 is running management tool stacks and device drivers such that Dom0 must run first when Xen boots and Xen is unusable without Dom0.

Guest domains (DomU) besides Dom0 have paravirtualized device drivers (PV drivers). The PV drivers consist of backend driver running in Dom0 and frontend driver running in DomU. The backend and frontend driver communicates to each other through ring buffers maintained by Xen. By leveraging Dom0's device drivers, the PV driver can exploit a wide range of devices that are not supported by Xen.

Xen supports HVM domain which is Hardware Virtualization Mode, or hardware-assisted virtualization. Unlike paravirtualized domain, HVM domain does not require guest operating systems to be modified. However, Xen HVM mode requires hardware support by the host CPU such as Intel VT-x and AMD AMD-V. HVM mode is known for naturally protection from the Meltdown vulnerability [67] because Meltdown is ineffective in a complete virtualization environment.

2.7 LibrettOS

LibrettOS [79] is a new OS design first introducing a fusion of multiserver OS and LibOS. The default mode of LibrettOS is multiserver OS running system services in an isolated manner. A network server is introduced as an example of system servers. It is built on a rumprun unikernel and contains the NetBSD's latest 10GbE NIC driver. The network server drives the NIC hardware to forward incoming and outgoing network packets to target applications. Therefore, applications can share the NIC hardware which is a limited resource. The multiserver OS mode brings isolation to the networking system such that any fault and security vulnerability potentially existed in the code can be isolated. For example, a deadlock that occurred in the network stack can be resolved by restarting the network server. Unlike

monolithic OSs, faults in system components do not affect the entire system.

For selected applications, LibrettOS also acts as a LibOS. When an application requires the better performance, it is granted exclusive access to hardware resources such as networking and storage. This can be feasible because the LibOS mode allows the application to contain the device driver code as a library in its address space.

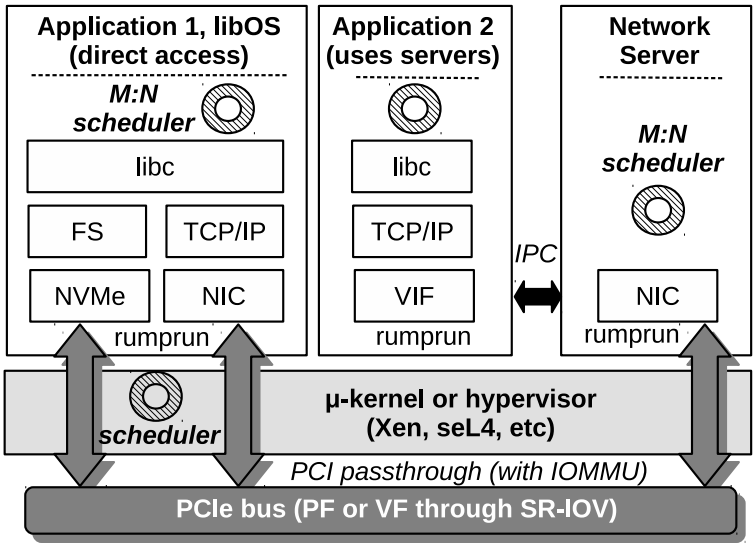


Figure 2.2: Design of LibrettOS.

Furthermore, LibrettOS has the ability to switch dynamically between two modes: applications can switch between multiserver OS and LibOS mode during the runtime with no interruption. In addition, LibrettOS can solve one of the critical limitations of LibOSs, which is limited driver support. As LibrettOS is based on rumprun unikernel, it can leverage NetBSD's device drivers.

LibrettOS is the first operating systems that can simultaneously address issues of isolation, performance, and recoverability while still using the existing NetBSD drivers and software. LibrettOS' design has a unique advantage in that, the two paradigms seamlessly coexist in the same OS, enabling end-users to simultaneously exploit their respective strengths (e.g., greater isolation, high performance).

Fig. 2.2 shows the design of LibrettOS, and the figure is reproduced here from [79] under fair use and is shown for completeness.

Chapter 3

Related Works

Researchers have proposed isolation on the various components of systems such as user/kernel, trusted/untrusted within kernel code or application code, and system component compartmentalization. Moreover, they have attempted to find a new way to impose isolation by using the software-based solution, leveraging new hardware features, or both. In this chapter, we discuss some related works about improving security with enhanced isolation.

3.1 presents unikernels which is used by this dissertation. Next, 3.2 discusses about software component isolation. Finally, 3.3 introduces various OS designs providing isolation of system components.

3.1 Unikernels

Since their invention in 2013 [68], unikernels have grown in popularity in academia. These single-purpose, minimal VMs offer benefits in addition to the already mentioned performance gains. They are lightweight [97], offering subsecond boot time and very low disk/memory footprints. This is due to the simplicity of unikernel LibOSs and the fact that in a unikernel instance, the kernel embeds only what is needed for the application it runs. Lower footprints translate into cost reductions for the cloud tenant and superior consolidation (increased revenue) for the provider. Fast boot times make unikernels good candidates for scale-out/elastic deployments [80]. The potential application domains for unikernels are plentiful, as listed in Chapter 1.

The isolation between unikernel instances running on a host is strong as they are virtual machines, and they are considered superior to containers [70] in that regard. However, in this dissertation, we show that the lack of intra-unikernel isolation is a security issue and addressed that concern. To our knowledge, we are the first to propose an intra-unikernel isolation system.

The performance benefits of unikernels come at least partially from the sharing of a single and unprotected address space [51, 59, 81]. That concept was originally pioneered by single-address-space OSs that appeared in the 90s following the appearance of 64-bit virtual addressing, such as Opal [18] or Nemesis [62]. We demonstrated that using a lightweight isolation mechanism such as MPK can bring security benefits while keeping a low latency for system calls.

Although some unikernels such as Rumprun [49], OSv [51], and HermitCore/HermiTux [59, 81] are entirely written in unsafe languages (C/C++), others use memory-safe languages. These include MirageOS [68] written in OCaml, LING [21] in Erlang, HaLVM [115] in Haskell, and RustyHermit [60] in Rust. However, even those rely on memory unsafe languages or unsafe code blocks to implement the low-level operations that an OS needs to support. Using our isolation scheme, we show that the safe part of the kernel can be isolated from the unsafe regions.

3.2 Software Component Isolation

Beyond the traditional user/kernel split, the decomposition of software into trusted and untrusted components have been studied in several past works, at the application [8, 58, 111] and OS [106, 109, 110, 114] levels. LibOSs such as Graphene [109, 110] adopt the Exokernel OS model and bring as many kernel components as possible in user space, reducing the size of the interface with the kernel for more isolation. In VPFS [114], the file system service

is split between two isolated components, a small and trusted computing base performing security-critical operations and an untrusted code base reusing most of the code of an existing legacy file system. In Proxos [106], the system call interface is partitioned into trusted and untrusted operations. Configuration rules allow routing the application's system calls either to a trusted microkernel or to an untrusted commodity OS. Occlum [96] runs a LibOS within an Intel SGX enclave and offers isolation for multiple tasks inside that enclave by leveraging Intel MPX [24] (deprecated in recent Intel CPUs).

Among the fine-grained isolation works focusing on the application level [8, 58, 111], SandCrust is relatively close to our work. It isolates safe from unsafe Rust code by running unsafe code in a separate process, which is not doable in a unikernel without breaking the single address space principle. To our knowledge, we are the first to apply fine-grained isolation to unikernels. Due to the peculiarities of this OS model, we face specific challenges such, as the need to keep a single address space to preserve a low system call latency and the need to reintroduce user/kernel space isolation.

3.3 OS Designs Providing Isolation

Microkernel model [5, 30, 39, 42, 45, 46, 65] provide isolation of system software components in separate address spaces and it is the key principle of the microkernel model. Microkernels [37, 43, 52] introduce stronger security and reliability guarantees by placing only essential system components (scheduling, memory management, IPC) in the kernel whereas the traditional monolithic OS design has all the system components in a single kernel. L4 [65] is a family of microkernels, which is known to be used for various purposes and a notable member of this family is seL4 [52], a formally verified microkernel. Multiserver OSs are a specialization of microkernels where OS components (e.g., network and storage stacks, device drivers) run in separate user processes known as servers. MINIX 3 [42], GNU Hurd [17],

Mach-US [104], and SawMill [36] are examples of multiserver OSs.

For various layers of the system software in a virtualized environment, such as hypervisor [98], management toolstack [22], and guest kernel [78], decomposition and isolation have proven beneficial. Security and reliability are particularly important in virtualized environments because they have multi-tenancy characteristics and the reliance on cloud computing has been growing for today's workloads. In the desktop domain, Qubes OS [92] leverages Xen to run applications in separate VMs and provides strong isolation for local desktop environments. Other microkernels [52, 103, 112] with a potentially small code base can benefit by imposing virtualization as well.

The storage server of LibrettOS' multiserver mode obtains the security and reliability benefits of microkernels by decoupling the application from system components such as drivers that are particularly prone to faults [38, 44] and vulnerabilities [19].

IX [12] and Arrakis [84] bypass traditional OS layers to improve network performance compared to commodity OSs. IX introduces its custom API and leverages the DPDK library [108] to directly access NICs. On the other hand, Arrakis supports POSIX and it is built on top of the Barrelfish OS [9], but its device driver support is limited.

Researchers tried to employ certain aspects of the multiserver design using existing monolithic OSs. For example, VirtuOS [78] employ a fault-tolerant multiserver design by leveraging virtualization to strongly isolate the Linux kernel components. They run storage and networking servers as service domains on top of Xen. Snap [72] implements a network server in Linux to improve performance and simplify system upgrades. Snap uses its own (non-TCP) protocol and cannot be easily integrated into existent applications. Therefore, all network device drivers are required to be re-implemented.

Researchers have also proposed the sidecore approach [35, 56] for I/O optimization in VMs.

This approach is to avoid VM exits and offload the I/O work to sidecores. Kuperman et al. [57] proposed to consolidate sidecores from different machines onto a single server, because this approach is wasteful when I/O activity reduces.

Exokernel [34] first proposed LibOS that is to make OS components as libraries and link them to applications. Nemesis [62] implemented a LibOS with an extremely lightweight kernel. Drawbridge [85], Graphene [109], and Graphene-SGX [110] adopt the LibOS model such that they benefit the security benefits of LibOS. Bascule [10] demonstrated OS-independent extensions for LibOSs. EbbRT [93] proposed a framework for building per-application LibOSs for performance.

Chapter 4

Design of Intra-Unikernel Isolation Technique

This chapter goes through the design of our intra-unikernel isolation technique. We follow the design objectives: (1) preservation of a *single address space*, (2) isolation of various memory areas, and (3) *negligible cost*.

In this chapter, Section 4.1 discusses about assumptions and thread model. Section 4.2 introduces data considered to isolate. Section 4.3 presents isolation with MPK. Section 4.4 explains unsafe kernel isolation. Lastly, Section 4.5 describes user application isolation.

4.1 Assumptions and Threat Model

We define a unikernel application to be a collection of software components, i.e. pieces of code. These are compiled and linked together to form a unikernel binary, executed at runtime on top of a hypervisor in a VM representing a unikernel instance. The software components can either be trusted or untrusted. We assume no vulnerability in trusted components, which in practice denotes the use of a memory-safe language or verification techniques for these components. We assume that untrusted components can contain memory vulnerabilities such as buffer overflows, which can be exploited by an attacker aiming at hijacking the unikernel's control flow, leaking or tampering with sensitive data, etc.

We assume a unikernel model in which the LibOS is mainly implemented in a memory-safe language, examples of which include MirageOS [68], RustyHermit [60], LING [21], as well as HaLVM [115]. A unikernel is composed of application and kernel code. In this dissertation we aim to provide user/kernel separation so we simply see the entire application as an untrusted component, independently of application-specific characteristics such as the language it is written in or the level of skill of the application's programmer. In addition, we divide the kernel code into trusted and untrusted components. Trusted kernel components represent pieces of code written with a memory-safe language, i.e., offering strong security guarantees. Untrusted kernel components correspond to code written either in memory-unsafe languages [21, 68, 115] or in unsafe Rust code blocks [60]. To summarize, a unikernel is composed of (1) untrusted application code, (2) untrusted kernel components, and (3) trusted kernel components.

We assume that there is no vulnerability in the trusted kernel code, as memory safety is also guaranteed by Rust compiler. We trust the hardware to behave correctly and assume that there are no side channels.

4.2 Data Considered to Isolate

We have a general security principle: *untrusted code should access only what it needs to operate correctly.* Listing 2 shows an example of unsafe kernel code in RustyHermit. The function `write_byte` in `kmsg_write_byte` stores the input `byte` on the `KMSG` buffer. As `write_byte` writes the input at the destination decided by a raw pointer, it should be called in an unsafe code block. In this example, `write_byte` accesses the `KMSG` buffer through the local variable `buffer`. Therefore, the call to `write_byte`, the buffer `KMSG`, and the variable `buffer` should all be isolated.

Kernel code is comprised of safe components and unsafe components. Isolating unsafe ker-

```
static mut KMSG: KmsgSection = KmsgSection {
  buffer: [0; KMSG_SIZE + 1],
};

pub fn kmsg_write_byte(byte: u8) {
  let index = BUFFER_INDEX.fetch_add(1, SeqCst);
  unsafe {
    let buffer = &mut KMSG.buffer[index % KMSG_SIZE];
    write_byte(buffer, byte);
  }
}
```

Listing 2: Example of unsafe kernel code.

nel functions and variables requires separate .data/.bss sections for static data, stacks for function calls, and heaps for dynamic memory allocation. Thus, we create an isolated data section, isolated stack, and isolated heap for the unsafe components. For the user/kernel isolation, we isolate all the sections of user memory by creating another isolated .data/.bss section, isolated stack, and isolated heap for the user application.

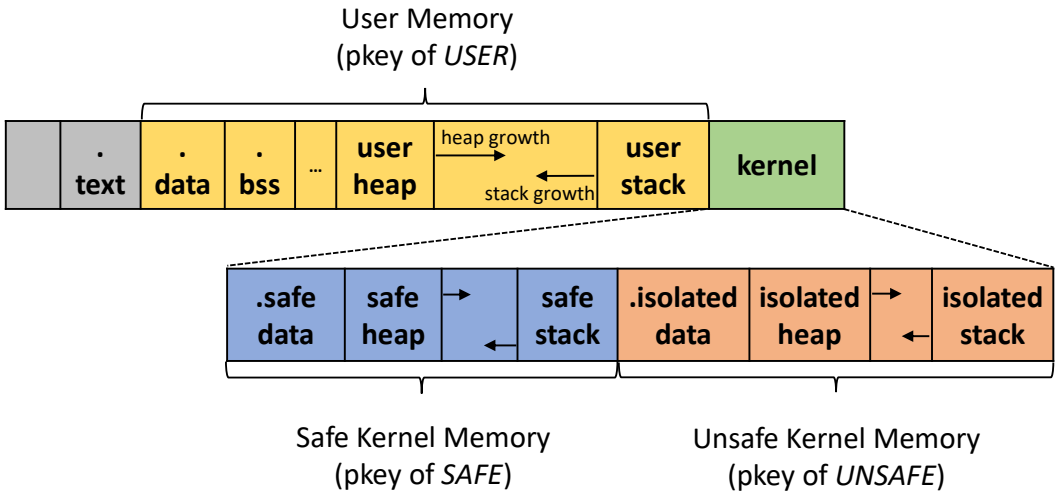


Figure 4.1: Virtual address space layout for intra-unikernel isolation.

Figure 4.1 shows the virtual address space layout of safe sections for the safe kernel com-

ponents, isolated sections for the unsafe kernel components, and user sections for the user application.

4.3 Isolation with MPK

We leverage Intel MPK for intra-unikernel isolation. As previously described, MPK provides per-thread permissions for groups of pages according to their protection keys (pkeys). We set a pkey of `UNSAFE` on pages of the isolated data section, stack, and heap. On the other hand, pages for the safe kernel memory sections have a pkey of `SAFE` and pkey of `USER` for the user memory.

```
unsafe {                        // pkey of SAFE is 1
    pkey_safe_NO_ACCESS();      // MPK_WRPKRU(0b0...01100)
    *ptr = some_data;           // Raw pointer dereference
    unsafe_function(ptr);       // Unsafe function call
    asm!("NOP"::::);            // Inline Assembly
    pkey_safe_READ_WRITE();     // MPK_WRPKRU(0b0...00000)
}
```

Listing 3: Example of isolating unsafe kernel code: raw pointer dereference, unsafe function call, and inline assembly. `MPK_WRPKRU` writes a value of 32bit on PKRU register.

We switch the current thread's permission for the pkey `SAFE` to "*No Access*" right before calling an unsafe function. Right after the function returns, the permission is switched back to "*Read Write*" to end the isolation. An example in Listing 3 shows that the permission for the `SAFE` memory region is set to *No Access* before executing the unsafe kernel code (raw pointer dereference, unsafe function call, and inline assembly) by setting a value of `0b0...01100` (`SAFE` pkey is 1: 2nd, 3rd bits are set to 1s for *No Access*) in the PKRU register. After the function returns, the permission is set back to *Read-Write* by writing a value of `0b0...00000` to the PKRU. Therefore, MPK prohibits the thread from accessing the `SAFE` memory region (whose PTEs contain the pkey of `SAFE`) while executing the `UNSAFE`

function. If a thread executing untrusted code (unsafe kernel or user application code) tries to access the safe memory region, a protection key page fault occurs and terminates process execution.

4.4 Unsafe Kernel Isolation

Unsafe code blocks are containing unsafe function calls, raw pointer dereference operations, and inline assembly in kernel code. Some of them need to access global variables or local variables in the stack frame of their caller function. Therefore, we create separate sections for static data isolation, a stack for unsafe function calls, and a heap for any dynamic memory allocation required by the unsafe functions. Those isolated memory regions are protected from the user application by MPK with the `UNSAFE` pkey.

Static Data Isolation. Unsafe functions in the kernel may need to access global variables. We define global variables accessed by the unsafe kernel code as *unsafe global variables*. We place the unsafe global variables into a separate memory section (`.isolated_data` section in Figure 4.1). On the other hand, global variables that are only used by safe kernel code should be located in the safe data section, which unsafe kernel code is not able to access. We minimize the number of global variables in the unsafe data section by including only those needed, so a compromised thread in the unsafe kernel code can only access a very limited part of memory.

In the real kernel code, some global variables are needed by both safe and unsafe code. We also put those shared global variables in the unsafe data section. As our objective is to minimize the number of global variables accessed by the unsafe code, all the rest of the global variables are protected by the unsafe code. Although having a separate .bss section for uninitialized global variables is useful to reduce the size of binary, we keep the variables

in the data section to ease design complexity while still attaining the reasonably small size of a unikernel.

Stack Isolation. An unsafe function should not share its function call stack frame with a safe function. We create a separate stack isolated by MPK pkey for unsafe functions, shown as `.isolated_stack` in Figure 4.1. When an unsafe function is called, we switch the value of the stack pointer register (`%rsp` in `x86-64`) with the address of the isolated stack.

By default, an unsafe function is strictly isolated, so it is unable to access the safe stack frames. In real kernel code, however, an unsafe function may try to access its caller's stack frames through local variables. If the caller is a safe function, the access should be managed carefully. In this case, we only allow access to the shared stack frame between the safe caller and the unsafe callee, meaning the unsafe callee function is still not able to access the rest of the caller's stack frames.

Heap Isolation. An isolated heap is required for unsafe code to allocate memory dynamically. We create a separate heap (isolated heap in Figure 4.1) and a memory allocation function (`unsafe_allocate`) for it. The `unsafe_allocate` function assigns available virtual and physical addresses and maps them while writing the pkey of `UNSAFE` to the corresponding page table entries. Consequently, a thread with inaccessible permissions for the safe memory region can still access the memory allocated by the `unsafe_allocate` function while executing the unsafe code.

4.5 User Application Isolation

The entire user part of the address space is assumed to be untrusted. For that reason, we separate the entire memory of the application from the kernel memory as the traditional monolithic kernel model does. However, separation is done by MPK, for which the do-

main switch operation, a simple update of the PKRU value, is much faster than traditional user/kernel separation methods involving costly world switch interrupts. Consequently, it fundamentally follows the main principle of unikernels: *a single address space*.

As the entire user application is treated as a set of untrusted components, all the memory sections are separated: .data/.bss, user stack, and user heap (Figure 4.1). A thread running a user application code should not be able to access either kernel memory regions, safe or unsafe. The border between user and kernel is quite distinct: a thread enters the kernel when system calls are called and exits the kernel when the system calls return.

User application memory also comprises user static data (.data, .bss, etc.), user stack, and user heap like those of the kernel. We can reuse most of the design choices used for safe/unsafe kernel isolation.

Chapter 5

Implementation of Intra-Unikernel Isolation Technique

We implement a prototype on top of RustyHermit to demonstrate our techniques. We can leverage Rust's features such as Rust Macros [88] to provide developers with a convenient way to use our isolation mechanism on the existing kernel source code.

In this chapter, Section 5.1 discusses about protection keys and mpk permission. Section 5.2 introduces unsafe kernel isolation. Section 5.3 presents copy between safe/unsafe kernel code. Section 5.4 explains user application isolation. Section 5.5 describes user bits on page table entry and applicability.

5.1 Protection Keys and MPK Permission

Isolating safe/unsafe and kernel/user memory requires two MPK protection keys. The protection key of 1 is used for the safe kernel memory region permission, while 2 is used for the unsafe kernel memory regions. As the user application is the most untrusted component, it is not protected by any protection key.

Table 5.1 summarizes PKRU values that determine permissions for the groups of pages by the protection keys. A thread running with a PKRU value of `0x00` is the most trusted entity at that point. However, when the thread executes an unsafe kernel code block, its PKRU

Table 5.1: PKRU values for memory regions: when a thread executes each code, PKRU is set to the corresponding value. For example, before a thread executes the user code, PKRU is set to contain `0x3C` (`No Access` on both safe and unsafe kernel memory regions) such that access to kernel memory by that thread is prohibited.

Memory Region	Unused 26 bits (pkey 3 ∼15)	`UNSAFE` (pkey 2)	`SAFE` (pkey 1)	Reserved (pkey 0)	Hex Value
Kernel (safe)	`0b00000000000000000000000000`	`00`	`00`	`00`	`0x00`
Kernel (unsafe)	`0b00000000000000000000000000`	`00`	`11`	`00`	`0xC`
User	`0b00000000000000000000000000`	`11`	`11`	`00`	`0x3C`

is set to contain `0xC` (`0b0000_1100`). This PKRU value prohibits the thread from accessing the group of pages of pkey 1, which corresponds to the safe memory regions. In the same way, `0x3C` (`0b0011_1100`) in the PKRU register prevents the thread from accessing both safe (pkey 1) and unsafe (pkey 2) kernel memory regions, providing the isolation of kernel from user memory.

5.2 Unsafe Kernel Isolation

Rust unsafe code [89] provides additional features such as raw pointer dereferences, inline assembly, Rust intrinsic functions, and unsafe function calls, as well as the use of static mutable global variables. As the Rust compiler does not guarantee memory safety in the unsafe code blocks, kernel developers should carefully use unsafe code at their own risk. However, all unsafe code can contain potential memory vulnerabilities. Accessing a static mutable global variable, for example, may expose a data race, but does not have memory vulnerabilities.

Rust Macro. Rust macros provide a handy way of reusing multiple lines of code [58]. As explained in Sections 4.3, 4.4, and 4.5, there are several steps involved in safe/unsafe and kernel/user isolation of global and local variables, and functions. All the procedures can be packed into an easy-to-use macro for better programmability. Listing 4

```
/********* Macro usage example *********/
unsafe_global_var!(
    static mut KMSG: KmsgSection = KmsgSection {
        buffer: [0; KMSG_SIZE + 1],
      }
    );

    unsafe fn write_byte<T>(buffer: *mut T, byte: T) {
      volatile_store(buffer, byte);
    }

    pub fn kmsg_write_byte(byte: u8) {
      let index = BUFFER_INDEX.fetch_add(1, SeqCst);
      unsafe {
        let buffer = &mut KMSG.buffer[index % KMSG_SIZE];
        isolate_function!(write_byte(buffer, byte));
      }
    }
```

Listing 4: Usage example of the macros in the kernel code.

provides an example of a macro that isolates an unsafe function introduced in Listing 2. Macro `isolated_function` wraps the unsafe function call and expands to multiple steps that isolate the function. For a global variable accessed by the unsafe function, macro `unsafe_global_var` locates the global variable in the isolated data section.

Macro `isolated_function` wraps the unsafe function call and performs: (1) switching the stack pointer to the isolated stack; (2) setting the MPK permission and writing it on PKRU register; (3) calling the unsafe function; (4) restoring the MPK permission; (5) finally restoring the stack pointer to the safe kernel stack. In addition, the unsafe function (`write_byte`) accesses the global variable (`KMSG`), so the global variable should be located in the isolated data section. We implement an `unsafe_global_var` macro which adds Rust attribute of `#[link_section = ".unsafe_data"]`. Listing 5 describes the definition of the rust macros.

```
1       /********* Macro definition *********/
2       macro_rule! unsafe_global_var! {
3           (static $name:ident: $var_type:ty = $val:expr) => {
4               #[link_section = ".unsafe_data"]
5               static $name: $var_type = $val;
6           };
7       }
```

Listing 5: The definitions of the rust macros used for isolating global variable.

Isolated Kernel Data Section. We wrote a linker script to specify the isolated data section (labeled `.unsafe_section`) at a certain address. When RustyHermit boots, the pkey of `UNSAFE` is set for the corresponding page table entries of the section. To allocate global variables in the `.isolated_data` section, we leverage Rust's attribute (`#[link_section]`) to dedicate variables to that specific section [87]. To ease use of that attribute, we provide the `unsafe_global_var` macro, which wraps the definition of a global variable with the `#[link_section]` attribute. Developers should explicitly wrap the definition of a global variable that is accessed by unsafe kernel code with the `unsafe_global_var` macro. Listing 4 shows how the global variable `KMSG` is wrapped with the `unsafe_global_var` macro (at line #2). The macro adds the attribute `#[link_section]` before the definition of the target global variable (line #4 in Listing 5).

Isolated Kernel Stack. We create a separate stack with the protection key of `UNSAFE` apart from the stack for safe kernel functions. This isolated stack is used when calling an unsafe kernel function such as `write_byte` in Figure 2. Switching the stack pointer for the unsafe function to use the isolated stack frame can be done by switching the value of `%rsp` register by the inline assembly. We provide a macro (`isolate_function` defined in Listing 6) to expand lines of inline assembly because isolating an unsafe function requires: (1) saving the current stack pointer; (2) switching the stack pointer to the isolated stack; (3) changing

```
/********* Macro definition *********/
macro_rule! isolate_function {
    ($f:ident($($x:tt)*)) => {{
        asm!("mov %rsp, $0;" // Store stack pointer
             "mov $1, %rsp;" // Switch to isolated stack
             "mov $2, %eax;" // N/A perm on SAFE memory
             "xor %ecx, %ecx;"
             "xor %edx, %edx;"
             "wrpkru;"       // Write %eax on PKRU
             "lfence"
        : "=r"(current_rsp)
        : "r"(isolated_stack),"r"(UNSAFE_PERMISSION)
        : "eax", "ecx", "edx" : "volatile");

        $f($($x)*);          // Actual function call

        asm!("mov $0, %eax;" // R/W perm on SAFE memory
             "xor %ecx, %ecx;"
             "xor %edx, %edx;"
             "wrpkru;"
             "lfence;"
             "mov $1, %rsp" // Restore stack pointer
        :: "r"(SAFE_PERMISSION),"r"(current_rsp)
        : "eax", "ecx", "edx" : "volatile");
    }};
}
```

Listing 6: The definitions of the rust macros used for isolating kernel function.

MPK permission to *No Access* on the safe kernel memory; (4) calling the unsafe function; (5) restoring the MPK permission to *Read Write* on the safe memory; and (6) restoring the stack pointer to the safe stack.

It only works, however, for an unsafe function that does not need to access its caller's stack frame. Some functions get references to local variables of the caller as function parameters and access them. To cover this case, we also provide a macro (`isolate_function_weak`) with extra steps for sharing the caller's stack frame. The macro that disallows accessing the

caller's stack frame is, by contrast, named `isolate_function_strong`. It is also possible that an unsafe function needs to access data in a frame of one of the caller functions (e.g., caller's caller and so on). We provide `share` and `unshare` macros for making local variables in the remote stack frames accessible/inaccessible to the unsafe function.

Placing annotations represents some effort on the programmer side. However, we consider it to be relatively low: in our effort to isolate RustyHermit safe/unsafe code and user/kernel space, less than 2% of the codebase was touched. It is also straightforward: a simple keyword to place. Finally, that process is guided: any overlooked variable will be identified at runtime with an MPK fault.

Isolated Kernel Heap. We create an isolated heap for unsafe functions to allocate memory dynamically. Instead of implementing a new memory allocation function for the isolated heap, we reuse the existing allocation function for the safe kernel heap. The memory allocation function maps a virtual-physical address by writing the physical address and page flags to the corresponding page table entries. The unsafe allocation function additionally sets a protection key of unsafe on the page table entries.

Raw Pointer Accesses, Inline Assembly. Dereferencing raw pointers and using inline assembly allows access to arbitrary locations in memory, so such techniques should be isolated in a way that does not change the stack. We thus implemented two macros for developers: `isolation_start` and `isolation_end`. The first macro, `isolation_start`, is used to indicate that the isolation starts, so it switches the MPK permission to *No Access* on the safe memory regions. The other one, `isolation_end`, is used to indicate the end of isolation, and it restores the MPK permission to *Read-Write*. Kernel developers should add `isolation_start` before a raw pointer deference or inline assembly to start isolation and `isolation_end` after them to finish isolation.

Non-isolated Function. There is a small amount of unsafe kernel code that cannot be isolated by our techniques. For example, the spinlock code has a few unsafe functions that are used by both safe and unsafe kernel code. Introducing isolation on the functions may cause deadlock. Functions such as `lgdt` or `load_cs` also cannot be isolated because they are called early in the boot process. We also do not isolate x86 I/O port instructions such as `in` and `out` because these functions manipulate device memory. Functions such as `wrmsr` and `rdmsr` are not isolated because they access machine-specific registers. It is worth noting that all of these unprotected unsafe code blocks are very small, most representing just a few instructions and extremely unlikely to represent vulnerabilities.

5.3 Copy between Safe/Unsafe Kernel Code

RustyHermit requires BIOS and boot loader data to be located in a fixed memory address. Accessing this data is done by unsafe functions because it is accessed via a raw pointer, and this data should also be isolated. However, we cannot apply our isolation mechanism to it, since RustyHermit stores it at a fixed address. To protect the data, we provide a copy mechanism. When a thread accesses the data (e.g., an eight-byte variable in a data structure), only eight bytes are copied to a per-core memory buffer (`unsafe_storage`). The thread then accesses `unsafe_storage` through an unsafe function. If the thread writes new data to `unsafe_storage`, it should be synced so it is copied back to the original data structure. These operations are protected by threads concurrently running on the other cores. This is because `unsafe_storage` is restricted to that core only by using `%gs`-relative addressing (i.e., each core contains a different base address in the `%gs` register).

The memory copy function is itself unsafe because it requires raw pointers for source and destination. We maintain a whitelist of memory addresses to limit arbitrary memory access by the copy function.

```
pub extern "C" fn sys_rand() -> u32 {
    return kernel_function!(__sys_rand());
}
```

Listing 7: A system call calling an internal function wrapped by the `kernel_function` macro.

The memory copy is, however, unsafe itself. Thus, we implement a safe memory copy function. Instead of using an intrinsic function (`copy_nonoverlapping`) directly, source and destination addresses are screened. Only addresses in a pre-registered whitelist can pass the address screening. The whitelist is a list of a minimum number of addresses that the developer registers with his care. As the whitelist must contain addresses as minimal as possible, the developer should carefully register addresses to the whitelist.

In addition to the unikernel-specific areas, per-core data is accessed by the copy mechanism. The per-core data is accessed by the unsafe functions (we introduce `get` and `set` methods presented in Listing 1), so it should be isolated. It is not suitable to locate the per-core data in the isolated data section because per-core data contains important data such as a pointer to the scheduler.

5.4 User Application Isolation

Isolating the user memory region is simpler than the unsafe kernel isolation because the application does not share global variables with the kernel. In consequence, their border is distinct and the MPK permissions should only be switched for system calls.

System Calls. System calls are the gate between user and kernel so MPK permissions and stack should be switched before making a system call and after returning from it. To avoid modifying the Rust standard library, we modified the definition of system call. Each system call calls internal calls (e.g., `sys_rand` calls `__sys_rand` in Listing 7) and the internal

function is wrapped with a `kernel_function` macro.

What the `kernel_function` macro does is similar to the `isolate_function` for unsafe kernel isolation. It expands into a few lines of inline assembly and switches the MPK permission and the stack pointer to the user stack.

Global Allocator. An application written in Rust obtains memory from the system at runtime through the Global Allocator [86]. We create a separate global allocator for the user application. As the kernel only uses the Rust global allocator in its initialization process, the pointer of the global allocator is switched and points to the user global allocator right before the application runtime starts.

5.5 User Bits on Page Table Entry and Applicability

As MPK is designed for user memory only, the protection key of a supervisor-mode address is ignored and does not control data accesses to the address [25]. However, unikernels have a single address space, so we had to set the `USER` flag on all the page table entries regardless of kernel and user space. The four-level page table is used by `x86-64`, so the `USER` flag should be set for all the page table entries in all levels of the page tables. We set the `USER` flag of the Level3 and Level2 page table entries from the hypervisor (uhyve) side and Level1 and Level0 from the RustyHermit side.

We believe our isolation mechanism and policies based on MPK are widely applicable in (1) other unikernels and (2) other OSs and applications written in Rust. Indeed, the proposed scheme is based on memory areas (i.e., data, stack, and heap) that are commonly present in programs and operating systems.

Chapter 6

Evaluation of Intra-Unikernel Isolation: Security

Unikernels such as RustyHermit are still an emerging technology and are not widely used in production. It was thus difficult to find known vulnerabilities we could use to validate our unikernel isolation scheme. As a result, we provide unikernel applications with handcrafted attack scenarios and demonstrate that our isolation technique successfully thwarts those attacks. We present 2 scenarios, respectively demonstrating (1) user vs. kernel space isolation and (2) safe and unsafe kernel code isolation.

Section 6.1 presents evaluating user vs kernel space isolation. Section 6.2 introduces security evaluation of unsafe kernel isolation. Lastly, Section 6.3 discusses about other attack scenarios.

6.1 User versus Kernel Space Isolation

In this scenario, we assume the application is external-facing (a web server, for example) and contains a memory corruption-based vulnerability that a remote attacker uses to perform arbitrary memory reads/writes. Examples are CVE-2013-2028 [2] for Nginx and CVE-2014-0226 [3] for Apache. In an unprotected unikernel, due to the lack of user/kernel isolation, the attacker would then be able to use the vulnerability to freely tamper with or leak sensitive kernel data. This could be used to break security mechanisms enforced by the kernel, such

```
!!! Isolation FAULT !!!
Faulty address: 0x823880 [kernel | data | RW]
Faulty PC:      0x402054 [user | code | RWX]
  -> /home/user/test/main.c:76
Registers:
- rax: 0x2
- rbx: 0x10040041A
...
```

Listing 8: Example of reported MPK fault.

as Address Space Layout Randomization (ASLR).

We reproduced this scenario by writing a simple unikernel application that accesses the kernel data segment. In an unprotected unikernel, an attacker could freely read and write kernel data. Our user application isolation scheme can prohibit this attack. As the user application is running with the MPK permission `USER`, which disallows to access the kernel memory (including the kernel data section). When the write operation is issued, an MPK fault occurs and unikernel execution is terminated. Our system also displays some information about the fault, such as the instruction pointer at the time and the faulty address, in order to help a system administrator investigate the attack. An example of the notification message printed on the screen for such a fault is presented in Listing 8

6.2 Unsafe Kernel Isolation

In this scenario, we assume that an attacker is able to hijack the control flow of the unikernel application and divert it to trigger the execution of buggy unsafe kernel code through a system call. Depending on the vulnerability in the kernel code, the attacker could then tamper/leak kernel data, escalate privileges, execute arbitrary code, etc. Examples of vulnerable kernel code called through system calls with specific parameters are numerous, with specific examples being CVE-2013-1763 [1] and CVE-2016-10229 [4].

We reproduced such a scenario by assuming an attacker is able to manipulate the parameters of the per-core kernel variable access methods presented in Listing 1. This would give the attacker arbitrary memory read/write capabilities. The safe/unsafe kernel isolation method we implemented prevents malicious calls to `set`/`get` methods from accessing memory that is not allowed, i.e. the majority of kernel memory. When the unsafe kernel code tries to access the inaccessible memory regions, an MPK fault terminates unikernel execution and provides the instruction pointer at that point as well as the faulty address.

6.3 Other Attack Scenarios

An attack scenario against our system would be unsafe code tampering with the PKRU. possible mitigation against such an attack would be to use binary analysis/rewriting to validate/sanitize any use of the WRPKRU instruction, as done in ERIM [111]. An attacker could also try to bypass such checks by using Return Oriented Programming (ROP) to jump to code snippets manipulating the PKRU. Classical mitigation used in all modern systems against ROP is ASLR. Although they are currently not implemented in RustyHermit, both static analysis and ASLR can be integrated without any runtime overhead.

There is also a possibility of information leaks or data-oriented attacks due to unused registers not being saved and scrubbed upon safe/unsafe code switches. We chose not to do so for performance reasons, as it is certain that saving and restoring registers, for example with the `xsaveopt` instruction, will increase the domain switch latency.

Chapter 7

Evaluation of Intra-Unikernel Isolation: Performance

We conducted a performance evaluation to demonstrate our design principles: providing isolation with minimal overhead. The objective of the performance evaluation is to answer the following questions: First, what are the overheads of switching across isolated safe and unsafe kernel code and across isolated kernel/user code? Second, what is the performance impact of such isolation on real applications? Third, how does our scheme perform in a multi-threaded environment? We chose vanilla RustyHermit as a baseline and compare our prototype against it. Our experimental setup has an Intel Xeon Silver 4110 CPU (2.10GHz, eight physical cores) with 64KB of L1 cache, 1024KB of L2 cache per core, and 11MB of L3 cache. The setup has 192GB of main memory and runs Ubuntu 18.04 with Linux 4.15 (needed for MPK support). Rust's cargo version is 1.40.0.

In this chapter, Section 7.1 discusses about performance evaluation of unsafe kernel isolation. Section 7.2 introduces performance evaluation of user application isolation. Lastly, Section 7.3 presents results of evaluation of real applications.

7.1 Unsafe Kernel Isolation

In this section, we evaluate the unsafe kernel isolation. We aim to measure the overhead of calling an unsafe kernel function isolated by our techniques. This is because isolating unsafe

kernel functions may contain the possible overhead (e.g., MPK permission switching, stack switching, data copying) compared to vanilla ones. We chose examples of some unsafe kernel functions and implemented a micro-benchmark to measure the time cost of the isolated unsafe kernel functions.

Write_bytes. `write_bytes` is an unsafe function writing `byte` to an arbitrary address. We isolate `write_bytes` with `isolate_function_strong` macros and call it one million times, then calculate the time cost of a single function call.

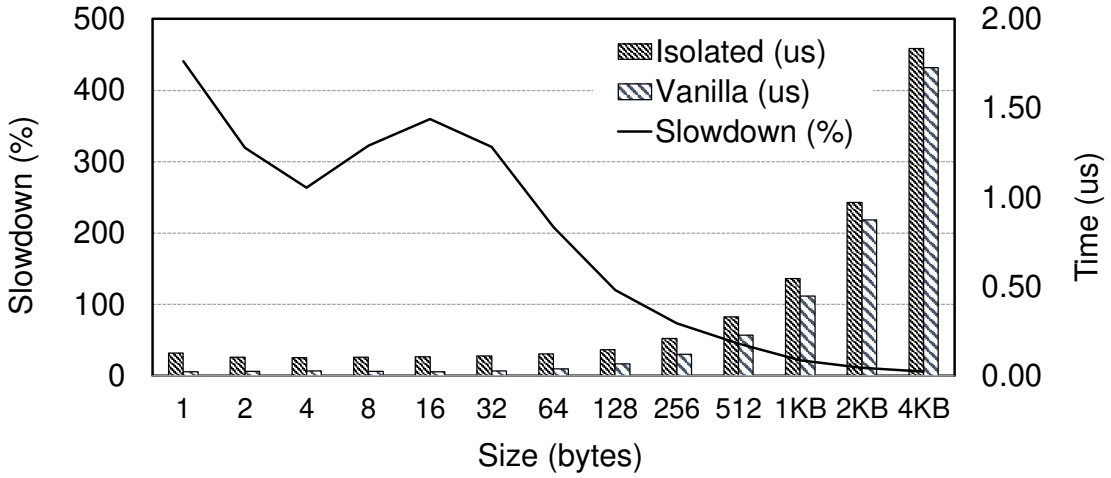


Figure 7.1: Cost of isolated `write_bytes` call.

The result, presented in Figure 7.1, contains the total cost of the unsafe function call, composed of: switching the kernel stack and the MPK permission, the actual function call, and restoring the stack and the MPK permission. We change the write size from 1 byte to 4KB. For each size, we iterate one million times and calculate the slowdown caused by the unsafe function isolation. With small writes, the isolated `write_bytes` is four times slower than the vanilla one. This is because the majority of the overhead comes from our isolation mechanism. However, as the write size increases, the cost of calling `write_bytes` dominates the overall cost and the isolation overhead becomes negligible. In particular, our prototype

introduces a 6% slowdown when writing 4KB at a time.

Per-core Variable Get and Set Methods. We also evaluated the cost of the `core_id` and `set_core_scheduler` functions to measure the per-core variable get and set methods (`Percore.get` and `Percore.set`). Introduced in Figure 1, `Percore.get` and `Percore.set` could be used as attack vectors to gain arbitrary memory read/write capabilities. Their usage as potential attack vectors means they should be isolated. In addition, they are invoked by kernel functions such as `core_id` and `set_core_scheduler`, which are frequently called in the kernel code. This makes them appealing as candidates for our unsafe kernel isolation as well. To do this, we created a micro-benchmark that iteratively calls `core_id` to invoke `Percore.get` and `set_core_scheduler` to invoke `Percore.set`. We measure the time cost of a hundred million calls, calculate the cost of one function call, and compare the isolated one to the vanilla one.

Table 7.1: PerCoreVariable `get` and `set` methods respectively called by `core_id` and `set_core_scheduler`.

Caller function	Unsafe function	Cost (µs)	
		Isolated	Vanilla
`core_id`	`Percore.get`	0.202	0.017
`set_core_scheduler`	`Percore.set`	0.367	0.020

Table 7.1 shows the results of the experiment. First, we observe the performance difference between `Percore.get` and `Percore.set` on both the isolated and the vanilla benchmarks. `set_core_scheduler` generally costs more than `core_id` because memory reads are faster than writes. When comparing the isolated functions to the vanilla ones, the isolated functions take longer due to the cost of memory copies introduced by the copy mechanism (Section 5.3): it introduces additional memory copy overhead besides the unsafe kernel isolation overhead (MPK permission switching, stack switching). `Percore.get/ set` copies the original per-core values to the unsafe memory region, which is followed by the unsafe read/write operation

(Listing 1) being performed on the unsafe memory region. Finally, the updated data is copied back to the original per-core data location. This additional overhead explains the performance degradation for the isolated `Percore.get/set` methods.

7.2 User Application Isolation

We evaluated user application isolation by measuring the cost of system calls, as they are a bridge between kernel and user space. To do so, we implemented micro-benchmarks written in both Rust and C and compare them. They exhibit null calls and `getpid` calls, the latter involving data copying. In addition to vanilla RustyHermit, we also evaluated system calls in Linux running on KVM. For Linux-KVM, we tested on an Ubuntu 17.10 distribution using Linux 4.13. We compiled all of the code with optimization level 3.

Null System Call. We evaluated a null system call to measure the pure system call latency. This call does nothing other than return, allowing us to measure the pure overhead of our user application isolation mechanism. For Linux, we use the `getpid` system call. We call this null system call a hundred million times and calculate the average cost for one function call. Note that we disabled vDSO for Linux in order to avoid potential user-mode system calls.

Figure 7.2A represents the cost of the null system call in the Rust and C applications. The isolated null system call in the Rust application takes 0.19 µs while the vanilla one takes 0.002 µs. This difference comes from the user application isolation mechanism that we provide. The vanilla system call only has the overhead of function call. However, the isolated system call introduces: (1) accessing the `Task` structure through the per-core scheduler (which can be accessed by `Percore.get` and also introduces the overhead mentioned in Section 7.1) to get the user stack address, (2) switching the MPK permission and stack pointer.

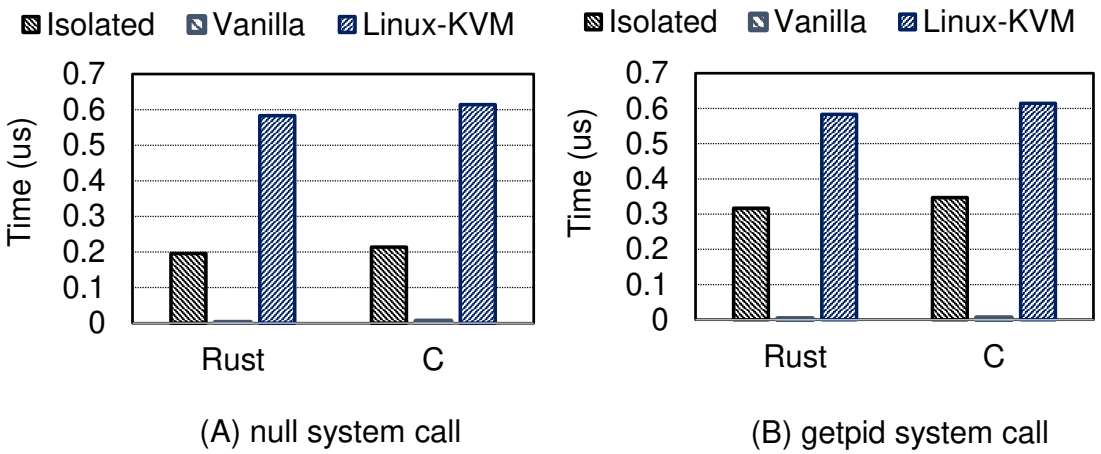


Figure 7.2: Evaluation of system calls.

Furthermore, the compiler loses optimization possibilities due to the use of the macros that we provide. Nonetheless, the system call isolated by the user application isolation mechanism is approximately three times faster than `getpid` on Linux (0.58 µs). This demonstrates that we can provide isolation while still maintaining the low system call latency feature of unikernels.

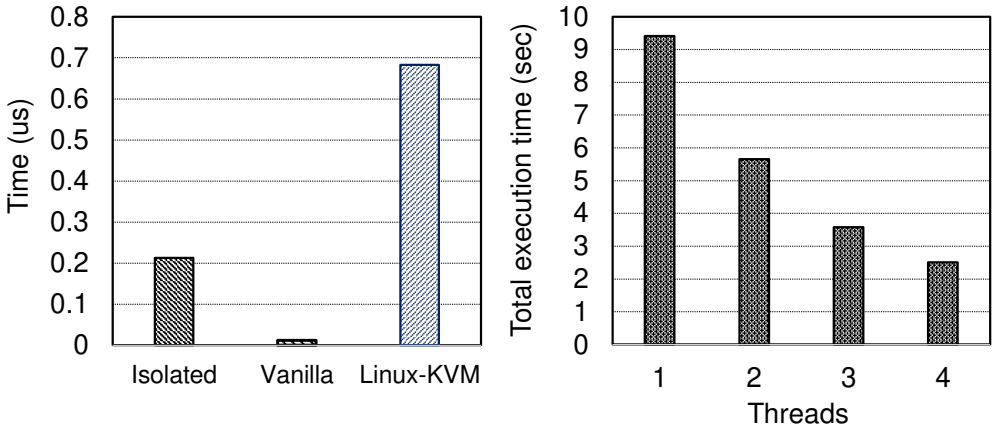
In the C application, all of the system call results are a bit slower (the isolated system call takes 0.21 µs, the vanilla version takes 0.005 µs, and the Linux version takes 0.61 µs). The user application isolation overhead still dominates the overall cost of the system call and reduces compiler optimization possibilities.

Getpid. This function is provided for user applications and invokes the `sys_getpid` system call. `sys_getpid` also contains unsafe/safe switches and the copy mechanism used for the per-core data. Thus, the cost of the `getpid` function can represent the overall overhead of the user application isolation mechanism. As in the null system call experiment, we set a micro-benchmark to make the call a hundred million times and calculate the average cost for one function call. Figure 7.2B presents the results of `getpid` on our prototype, vanilla

RustyHermit, and Linux. We tested both Rust and C applications.

In all cases, the system call from the Rust application outperforms that of the C application, as with the null system calls. In addition, the cost gap between Rust and C are similar to that for the null system call. The memory copy overhead is the main factor in the performance degradation of our prototype, as the PID of the task is stored in the `Task` structure that is referenced by the `current` pointer in the per-core scheduler. Accessing the per-core scheduler is performed via `Percore.get`, which introduces the additional memory copy.

With our scheme, the `getpid` system call is still 2x faster than it is on Linux, demonstrating that our technique preserves unikernel benefits.



(a) Time cost of sbrk (b) Multi-threaded getpid

Figure 7.3: Evaluation of sbrk and multi-threaded getpid.

Sbrk. We measured `sbrk` (only used by C applications) latency for evaluation of the user application isolation. We call `sbrk` with a parameter of 16 (an increment of 16). `sbrk` calls `sys_sbrk`, which does not include expensive per-core variable methods such as `Percore.get` and `Percore.get`. However, our user application isolation introduces the overhead of the MPK switch and the stack switch. Despite this, `sbrk` with our user application isolation still outperforms the Linux one significantly, as shown in Figure 7.3a.

Multi-threading. To demonstrate that our intra-unikernel isolation method works in multi-threaded environments, we created a Rust benchmark launching up to 8 threads and parallelizing an iteration of ten million `getpid` calls. We could observe that our intra-unikernel worked with multi-threading and scaled with the number of threads (Figure 7.3b).

7.3 Results on Real Applications

To measure the overall performance impact of our system, we evaluated our prototype with macro-benchmarks. We used memory/compute intensive benchmarks from various suites including NPB [94], PARSEC [13], and Phoenix [91].[1]

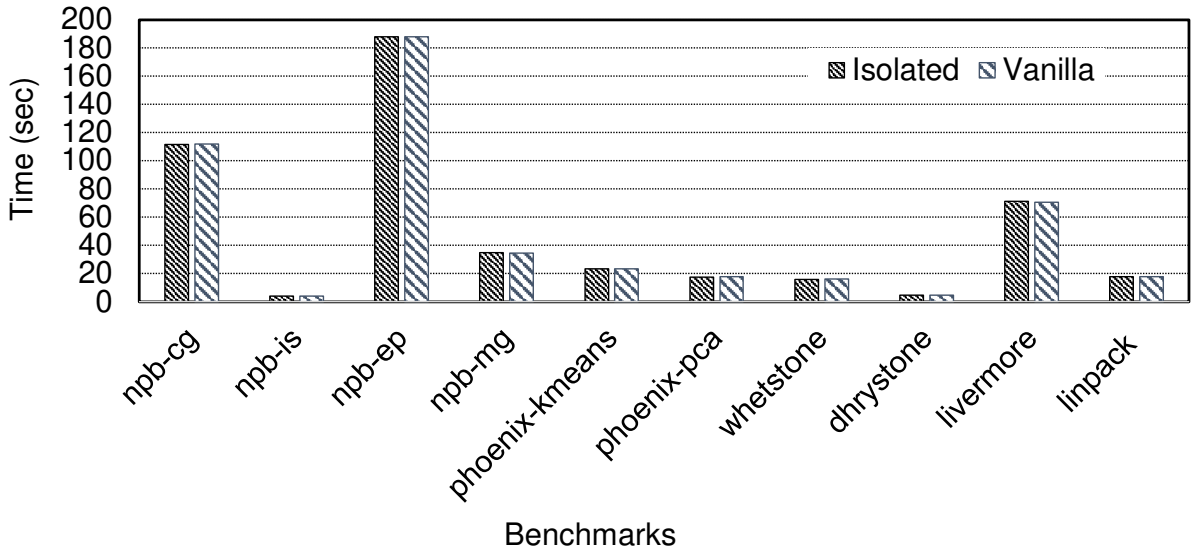


Figure 7.4: Execution times of macro benchmarks.

The results are shown in Figure 7.4, illustrating that the average slowdown imposed by the intra-unikernel isolation compared with the vanilla unikernel is only 0.6%.

We also counted the number of unsafe/safe switches and user/kernel switches and summarize them in Table 7.2. Remember that one unsafe function call corresponds to two unsafe/safe

[1]Note that some applications from these suites are not supported due to the limited compatibility of RustyHermit.

Table 7.2: Number of unsafe/safe switches and user/kernel switches invoked by benchmarks.

Benchmark	Unsafe/safe switches	User/kernel switches
npb-cg	5218	272
npb-is	4294	106
npb-ep	4370	116
npb-mg	4606	158
phoenix-kmeans	6882	1580
phoenix-pca	19402	7844
whetstone	3758	14
dhrystone	3734	10
livermore	13118	1574
linpack	3878	38

switches (from safe to unsafe switch on entry and unsafe to safe switch on return) and one system call corresponds to two user/kernel switches. Especially, phoenix-pca has a total of 27,246 switches and switches at a rate of 1,238 per second, which is system intensive. The evaluation demonstrates that our system introduces negligible performance overhead for real applications.

Chapter 8

Design of Storage Server for LibrettOS

In this chapter, we describe the design of the storage server for LibrettOS. We target to add strong isolation on the system components in order to enhance security, but still achieve reasonable performance. Section 8.1 describes the overall architecture of the storage server and applications. Section 8.2 and Section 8.3 present the frontend and backend drivers respectively. Section 8.4 discusses initialization process using hypercalls. Our IPC design is explained in Section 8.5. Lastly, Section 8.6 describes NVMe and the number of memory copies.

8.1 Storage Server and Application

Figure 8.1 depicts the architecture of storage server. Storage server and applications are built upon HVM mode rumprun unikernels on Xen hypervisor. The frontend driver is linked as a library in the application's address space in order that the application can communicate to the storage server. On the storage server-side, the backend driver is linked as a library along with other device drivers such as the NVMe driver. The frontend and backend drivers communicate through our own IPC. The detailed design of the IPC is discussed in Section 8.5. Applications can issue block I/O requests through the frontend driver. Through the IPC channel, the I/O requests are transferred to the backend driver. The backend driver can

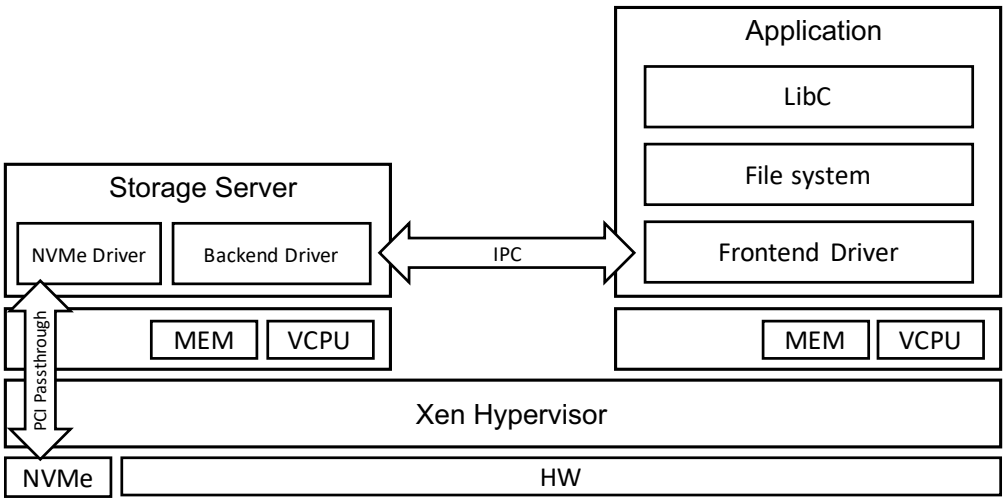


Figure 8.1: Design of the storage server.

request the receiving I/O to the actual storage device drivers (e.g., NVMe driver). As described in Figure 8.1, the device driver can exclusively access the hardware resources due to Xen's PCI-passthrough feature.

On the application domain, the frontend driver creates a virtual block device node such that the application can mount, open, write, and read on the node. The frontend driver receives `struct buf` from the file system on the upper layer. `struct buf` is the buffer header of NetBSD. The frontend driver transfers the data and metadata selectively chosen. From careful consideration, we set the size of block IO to 64KB. For the larger data I/O (e.g., 2MB), the file system divides the data into 64KB segments, creates several `buf`s, and issues block I/O. As we run storage server and applications on each rumprun unikernel, strong isolation is provided by virtualization.

8.2 Frontend Driver

The frontend driver is located in an application domain. Its fundamental role is sending and receiving data to be written/read from the storage device or ramdisk. The frontend transfers

data through the ring buffers and also initializes them. The frontend driver allocates a chunk of memory and grants references of the pages for the backend driver such that the backend driver can map the pages to its address space. As we opted to run our servers and applications as Xen domains, the frontend driver leverages Xen's APIs to share its memory and send interrupts.

The frontend driver initializes six rings and two data buffers. Each ring is coupled with atomic variables which are used to notify the other end about its status (e.g, asleep or awake).

The frontend driver creates a dedicated thread to receive responses from the backend driver. The thread is called the receiver thread. The receiver goes to sleep right after it is created. When the backend driver sends a response for the I/O request from the frontend, it also wakes up the receiver thread by sending a VIRQ. The receiver thread checks entries from the ring buffer and consumes them. It copies the data to the corresponding buf in case of a read operation and calls `biodone` to finish the I/O operation.

The frontend driver creates a virtual block device node at `/dev` directory. The device node is an interface for applications to do block I/O. Like a regular file or block device, an application can open, read, or write data on it. The application can mount the virtual device node on its file system as it does with a regular block device. We build the frontend driver as a library such that applications can link it in their address space.

8.3 Backend Driver

The backend driver is located in the storage server domain and its role is forwarding block I/O requests to the storage device and replies responses back to the frontend driver. As explained in the previous section, the backend maps the shared pages in its address space

using Xen's APIs and grant references. Similar to the frontend's receiver thread, there is a dedicated thread for receiving. The receiver thread of the backend driver is created when the communication channel (ring buffers and VIRQs) is built. After the thread is created, it also goes to sleep and wakes up by the VIRQ from the frontend.

To build the communication channel, the backend and frontend drivers need to provide each one's information required for mapping the shared pages and building event channels. Provision of the information is done by hypercalls. We introduce a few hypercalls for the frontend and backend drivers. After they are initialized, they call hypercalls to register their information in Xen's memory. More details are described in Section 8.4.

8.4 Driver Initialization and Hypercalls

The backend driver must be running before the frontend driver starts. The backend driver allocates a welcome port. Then it calls a hypercall to register the port in Xen's memory. The welcome port is an event channel port that can be bound to the frontend driver's hello port. When a frontend driver starts, it initializes shared pages, builds ring buffers, their atomic variables, and allocates the main event channel ports which are used as the VIRQs. Then the frontend driver retrieves the information of the backend driver via a hypercall. The information includes the domain id number of the domain where the backend driver is running, and the welcome port that the backend driver allocated before. With the information, the frontend driver can construct an event channel. After the first event channel is built, the frontend driver registers its information in Xen's memory through a hypercall. Finally, the frontend driver pokes the backend driver by sending a VIRQ on the welcome port of the backend, or the hello port of the frontend. The backend driver's handler of the welcome port executes the process of connecting with the frontend driver (mapping the shared pages, binding the main event channel for VIRQs).

8.5 Inter-Process Communication

The communication channel between the backend and frontend drivers consists of ring buffers and VIRQs. In our design, six rings and two data buffers are created on the frontend's memory and shared with the backend driver (see Figure 8.2). Two VIRQs are used for signaling the other end. We leverage Xen's grant references for the shared pages and Xen's event channels for the VIRQs. To avoid performance degradation of the hypercalls, we preallocate shared pages and VIRQs on the frontend's initialization stage. Once the shared pages and VIRQs are built, there is no additional allocation such that additional hypercalls are avoided.

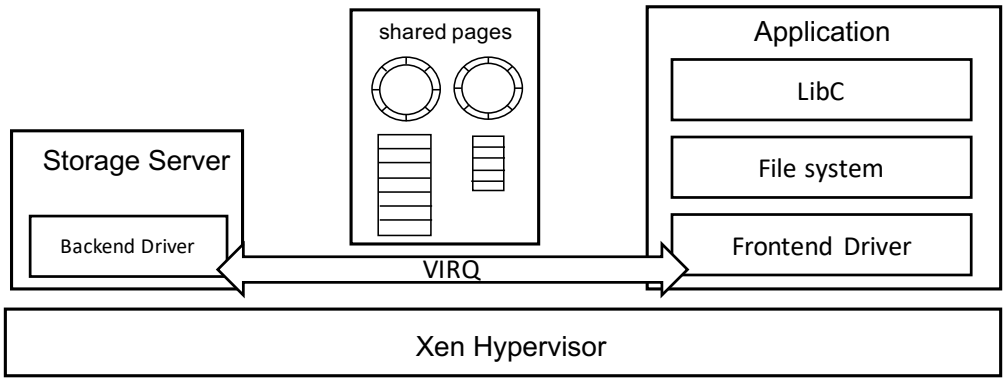


Figure 8.2: Illustration of inter-process communication (IPC).

Nonetheless, sending a VIRQ requires a hypercall. Each time one end produces an entry in the ring buffer, it should ping the other end to consume the entry. Sending VIRQ costs the overhead of the hypercall, so we need to minimize the number of the VIRQ sent during the I/O process. To avoid frequent VIRQs, we create atomic variables and attach them to the ring buffers. When one end is consuming the ring buffer, it sets a value on the corresponding atomic variable attached to the ring. With the value in the atomic variable, the other end (producer) can know the consumer is running. Therefore, the producer doesn't need to send VIRQ to the consumer. If the producer knows the consumer is sleeping, it sends

VIRQ to wake the consumer to start consumption. Both backend and frontend drivers are either producer and consumer in each case. When sending write requests from the frontend, the frontend is the producer of the `write request allocate ring` and the backend is the consumer. When sending the I/O responses, the backend is the producer of the `write response allocate ring`, and the frontend is the consumer.

Ring Buffers. The ring buffers are key components of our IPC design. We use a lock-free ring buffer implementation from [77] There are total of six rings and two data buffers. Two free rings (large free ring and small free ring) are the list of available indices of the two data buffers (large buffer and small buffer respectively). For requesting to write, the frontend driver needs an entry of the large data buffer because the write request contains data to be written. On the other hand, For requesting to read, the frontend driver uses an entry of the small data buffer because the read request only contains metadata such as the size of data read and block number of the storage device. The backend driver creates responses of the write operation with the small buffer because the write response contains the size of data written and error code. For responses of reading, it creates responses with a large buffer, because the read data is transferred through the responses.

The ring buffers are built on the shared pages between the frontend and backend drivers. The frontend initializes the ring buffers and the shared pages only when it is launched. Therefore, there is no additional overhead from the hypercalls during the runtime.

Virtual Interrupt. Interrupt is also one of the key components of the IPC. As the backend and frontend drivers are running in each guest domain, we use Xen's event channel. There are three event channels built between the backend and frontend drivers. The first one is the welcome channel which is discussed in Section 8.4. The other two channels are used for the data transfer. Sending a VIRQ through the event channel requires to call a hypercall.

To avoid excessive hypercall overhead, the sender should not send VIRQ while the receiver is active. We use atomic variables for this purpose. Each atomic variable is attached to each ring and shared by both ends. When the receiver thread is looking at the ring, it sets the atomic variable with a dedicated value in order that the sender knows. With the atomic variables, unnecessary VIRQs are avoided and this leads to minimizing the number of calls of the hypercall.

8.6 NVMe and Memory Copies

The NVMe driver is linked as a library in the unikernel image in order that the storage server directly accesses the NVMe device. We implement a glue code to build and link the NVMe driver. Furthermore, the NVMe device is enumerated in the rumprun unikernel by Xen's PCI-passthrough. The backend driver creates a `buf` structure and fills out the meta data. For a write operation, we opt to reuse the data buffer of the ring buffer. The buffer pointer in the `buf` points to the data buffer in the ring buffers to avoid an additional copy. For a read operation, on the other hand, a new memory buffer is allocated and data read from the NVMe device is written on it. By reusing the data buffer of the ring buffer, we could limit the number of memory copies. In the current design, there are two copies for read and one copy for write.

Chapter 9

Implementation of Storage Server

In this chapter, we discuss the detailed implementation of the design of the storage server described in Chapter 8. Section 9.1 discusses about rumprun HVM mode. Section 9.2 introduces modification on Xen hypervisor. Section 9.3 presents frontend driver. Section 9.4 explains backend driver. Section 9.5 explains ring buffers and virtual interrupt. Lastly, Section 9.6 describes block I/O routine.

9.1 Rumprun HVM Mode

Our storage server and applications are built on Rumprun HVM mode. Since unikernel is required to have application code, we need to code a simple application code for the storage server. We put an infinite loop doing nothing but sleep in the application code.

The backend and frontend drivers are linked to applications as libraries. We create a library called librumpdev_myblk, that is compiled with the backend or frontend driver code for each sake. We build rumprun unikernels with NetBSD 9.0.

9.2 Modification on Xen Hypervisor

We also make a modification on Xen hypervisor. Our custom Xen hypervisor is based on Xen 4.14 and include a new hypercall that is used in the domains' initialization. Our new hypercall has operations: (1) registration of the backend and frontend driver; (2) fetching

the information of each driver. The registration operations of the hypercall simply store the information of the drivers (e.g., domain id, port numbers, and grant references) in Xen's memory. On the other hand, the fetching operations copy the information from the Xen's memory to each driver's memory. Once the backend and frontend drivers are connected through their communication channel (i.e., IPC), our custom hypercall is not used.

9.3 Frontend Driver

The frontend driver is in charge of bridging applications and the storage server. In addition, it allocates pages to be shared with the backend driver and creates ring buffers in the shared pages. It also initializes the atomic variables attached to the rings.

Initialization of Ring Buffers. Right after the application domain is launched, the frontend driver starts the initialization process. It queries the information of the backend driver via the hypercall. Then, it allocates memory and creates rings and data buffers. Six rings are created and two data buffers are allocated in continuous pages. In the `frontend_init_ring` function, four pages are allocated. These four pages contain the grant references of the shared pages. The grant references of the four pages are stored in Xen memory through the hypercall. Later, the four pages are mapped by the backend driver such that it can get the grant references for the shared pages. The atomic variables attached to the rings are initialized too. After the ring buffers are initiated, the frontend driver allocates the event channel ports and register itself through the hypercall. Finally, a VIRQ is sent to the welcome port of the backend driver.

Virtual Block Device and Driver Library. The virtual block device node is an interface that applications can communicate with the frontend driver. We code a driver library that creates a virtual block device node in /dev directory. Also, the library defines function

wrappers such as open, read, write, and ioctl. File operations called by the application are routed to the functions in the driver library. All the backend and frontend drivers' code is also linked to the driver library. Therefore, both the storage server and application domain should link the driver library to their applications.

Data Transfer and Status Synchronization. Data of the block I/O is transferred through the ring buffers. When the application requests block I/O, the pointer of the `struct buf` is passed to the frontend driver. The frontend driver extracts the data, size of data, and block number from `b_data`, `b_bcount`, and `b_blkno` of the `struct buf` respectively. Then it copies the data, size, block number, and pointer of the `struct buf` to the ring buffer. It has to track the pointer of `struct buf` in order to close the I/O request with `biodone` function. Finally, it sends VIRQ to the backend driver if it is asleep. More details are described in Section 9.6.

Threads that dequeue an entry from the rings go to sleep when the rings are empty and set the atomic variable to a negative value. As the atomic variables are shared, the backend driver knows whether the threads sleep or not. Later, they are wakened by the VIRQs from the backend driver. If the rings are not empty and the threads are working on the rings, the atomic value is set to 1 such that the backend driver does not send the VIRQs.

Closing Block I/O. The frontend driver is in charge of calling `biodone` function to close the block I/O request. The response from the backend driver contains the pointer of `struct buf`, size of data written/read, and error code. The frontend driver calculates the size of the remaining data to be processed and stores it in `b_resid` of the `struct buf`. Eventually, the frontend calls `biodone` with the buffer header (`struct buf`) containing all the metadata. At this point, the block I/O ends.

9.4 Backend Driver

The backend driver resides in the storage server domain. Its role is receiving I/O requests from the frontend, do I/O, and send responses back. As the storage server domain contains device drivers, the backend driver can directly send the requests to the device drivers. Similar to the frontend driver, the backend driver is built as a library and linked to the application in the storage server domain.

Initialization and Connecting to Frontend Driver The backend driver has to start before applications are launched. It first allocates the welcome port and registers its domain id, and the welcome port number. After the frontend driver finishes its initialization, a VIRQ on the welcome port is received and invokes the backend driver to connect with it. The grant references for the shared pages and event channel ports are already in the memory of Xen, the backend driver maps the shared pages and builds the event channels.

Receiver Thread. The receiver thread is also created when the backend driver is connecting with the frontend. The thread sleeps when the rings are empty, but wakes up by the VIRQ from the frontend driver. Rump kernel context should be obtained for the receiver thread because it calls kernel APIs related to the block I/O.

Ramdisk and NVMe. We also implement a ramdisk in the backend driver. 512 MB of memory chunk is allocated from the driver library code. To measure the pure overhead of the backend and frontend drivers, we use the ramdisk instead of real storage devices. This way, we can avoid the overhead that comes from the storage device.

As explained in Section 8.6, the backend driver creates an empty `struct buf` by `getiobuf`. then it fills out the metadata: `b_bcount` is the size of data, `b_blkno` is the block number, `b_dev` is the device number of the block device, `b_iodone` is the pointer to the callback

function, and `b_private` is for private use. We use `b_private` to store the address of `buf` in the frontend to track the I/O requested by the application. Also `b_cflags` is marked as `BC_BUSY` indicating that the `buf` is in use. The `b_flags` is set to `B_WRITE` or `B_READ` accordingly. For a write operation, the `b_data` points to the data buffer of the ring buffer to avoid a memory copy. For a read operation, a memory buffer is allocated and the pointer of it is stored in the `b_data`. Finally, the backend driver calls `bdev_strategy` with the `bp` parameter (pointer to the `buf` structure) to issue storage I/O to the NVMe device.

9.5 Ring Buffers and Virtual Interrupt

Ring Buffer. Ring buffers are built on the shared pages. To share pages between domains, the owner of the memory grants (frontend driver in our case) access to the other end who maps the pages. The frontend driver initializes the grant table in its early stage. Then, it sends the grant references to the backend driver. The backend driver maps the pages by calling `HYPERVISOR_grant_table_op` with the `GNTTABOP_map_grant_ref` parameter and grant references. The drivers pre-allocate a big chunk of shared memory on their initialization process, therefore they do not call the hypercall for a single I/O request as the existing blkfront driver does in Linux or rumprun.

The size of the data block and the total number of entries of the rings significantly affect the performance of the ring buffers. We are able to set up 2048 entries for each ring because more than 2048 is not allowed due to the limited memory. Also, we set the data block size to 4098 which is equivalent to the block size that NetBSD kernel is using.

VIRQ. We leverage Xen event channel to implement virtual interrupt between drivers. To allocate an event channel and the port of it, `HYPERVISOR_event_channel_op` with the `EVTCHNOP_alloc_unbound` parameter is used. Once the port number is allocated by Xen,

Table 9.1: Values of the synchronization variables.

	frings	arings
Active	1	1
Sleep	-2	0

the other end has to know it to bind. With the port number, one calls `HYPERVISOR_event_channel_op` with the `EVTCHNOP_bind_interdomain` parameter to bind it. Then, a new port number for the one binding is allocated by Xen. Each end has its own port number. To send a signal, one can call `HYPERVISOR_event_channel_op` with the `EVTCHNOP_send` parameter and port number that it has.

Values of Atomic Variables We create six atomic variables attached to six rings, but we only use four of them. The atomic variables of two frings are initialized to -2 and others to 0 when the rings are initialized. Two atomic variables of the frings with -2 are meaning the threads are sleeping. When they are active, the atomic variables are set to 1. The other atomic variables have 0 when the threads are inactive, and 1 for when they are active. The values of the atomic variables according to the status of the threads are shown in Table 9.1.

9.6 Block I/O Routine

The backend and frontend drivers communicate through ring buffers which are in the shared pages. We create six ring buffers, a large data free ring (`large_fring`), small data free ring (`small_fring`), write request allocate ring (`write_req_aring`), write response allocate ring (`write_rsp_aring`), read request allocate ring (`read_req_aring`), and read response allocate ring (`read_rsp_aring`). Two frings are maintaining available indices of data buffers.

Figure 8.1 describes a write operation: (1) The frontend driver dequeues an index from the large fring to (2) create a write request. (3) The data to be written is copied on the corresponding data buffer and (4) the index is enqueued into the write req aring. The

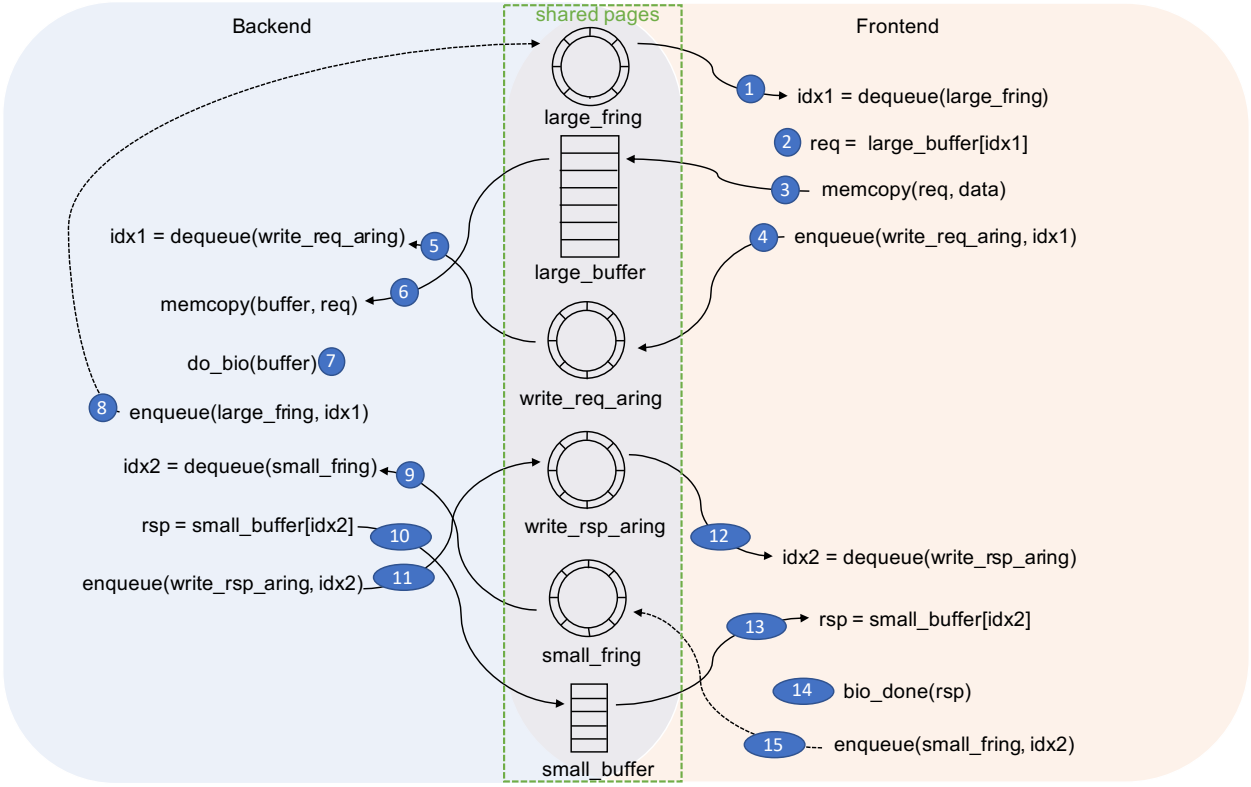


Figure 9.1: Block I/O routine of storage server.

frontend driver sends a virtual interrupt (VIRQ) if the other end (backend driver) is asleep, otherwise does not. (5) The backend dequeues the index of the data buffer containing the data from the write req aring. (6) The backend can do (7) a block I/O to real storage hardware or ramdisk. (8) The data buffer is free to be released, so its index is enqueued back to the large fring. (9) After the I/O is done, the backend driver creates a response by retrieving an index of the small data buffer. This can be done by dequeuing from the small fring. (10) The backend driver fills out the response with information (e.g., the original buf structure pointer, size of data written, and error code) and (11) enqueues the respond to the write rsp aring. Then, the backend driver sends a VIRQ if the frontend driver is blocked, otherwise does not. (12) The frontend driver dequeues the index from the write rsp aring and (13) accesses the rsp. (14) biodone (block I/O done) function is called. (15) Finally, the

frontend driver enqueues the released index to the small fring.

Chapter 10

Evaluation of Storage Server

In this chapter, we evaluate the storage server using micro- and macrobenchmarks and compare it against Linux. I/O from an application to storage goes through several layers such as file system, block layer, buffer cache, and so on. To measure overhead introduced by the storage server, we implement a microbenchmark and ran it with a ramdisk. Furthermore, we evaluate the storage server with an NVMe device. We also measure the performance of real applications with macrobenchmarks.

Section 10.1 explains our experimental setup. Section 10.2 presents the microbenchmark. Lastly, Section 10.3 discusses about the macrobenchmarks.

10.1 Experimental Setup

Table 10.1 shows our experimental setup.

Table 10.1: Experimental setup.

Processor	2 x Intel Xeon Silver 4114, 2.20GHz
Number of cores	10 per processor, per NUMA node
HyperThreading	OFF (2 per core)
TurboBoost	OFF
L1/L2 cache	64 KB / 1024 KB per core
L3 cache	14080 KB
Main Memory	96 GB
Network	Intel x520-2 10GbE (82599ES)
Storage	Samsung 970 EVO Plus 500GB NVMe SSD

We run Xen 4.14.0, and Ubuntu 20.04 with Linux 5.4.0 as Xen's Dom0 (for system initialization only). We use the same version of Linux as our baseline. Rumprun unikernel that we used for the experimental is the version maintained in the repository [102] and comes with NetBSD 9.0 code. We set the I/O block size to 64KB for the best performance. Also, the size of the ring buffer is set to 512 entries, built on 12 pages for a single ring.

10.2 Microbenchmarks

We implemented a microbenchmark repeatedly calling `read` and `write`. It first opens the virtual block device (/dev/myblk) as a single file. The block device is introduced in Section 9.3. We set the size of the message written to a file from 4KB to 4MB. The elapsed time of the iteration is measured by `my_gettime` function for the rumprun. `my_gettime` function can be called in the application code, and its function pointer is linked to a function (`frontend_gettime`) in the frontend driver. `frontend_gettime` function calls `rumpuser_clock_gettime` with the `RUMPUSER_CLOCK_ABSMONO` parameter for a timestamp. `rumpuser_clock_gettime` is a function that rumprun unikernel uses to get a value of monotonic clock in nanoseconds. Microbenchmark for Linux leverages the regular `clock_gettime` function for a timestamp. With the elapsed time, we calculate the average time for a single read/write call.

10.2.1 Ramdisk

We evaluate the storage server with the microbenchmark and ramdisk. For each size of the message, we measure the elapsed time of 100,000 iterations of read/write operation. However, we could only measure the iteration of 10,000 due to the limited size of the ramdisk in Linux.

We measured zero-copy and no-IPC versions as well. The zero-copy version comments out the memory copies in the backend and frontend driver code. In addition, the no-IPC version

excludes all the backend and frontend code. That is, the block device operation does nothing but returns with a return value of the successful block I/O. We could break down the overhead of the storage server by comparing the zero-copy version, no-IPC version, and the intact storage server.

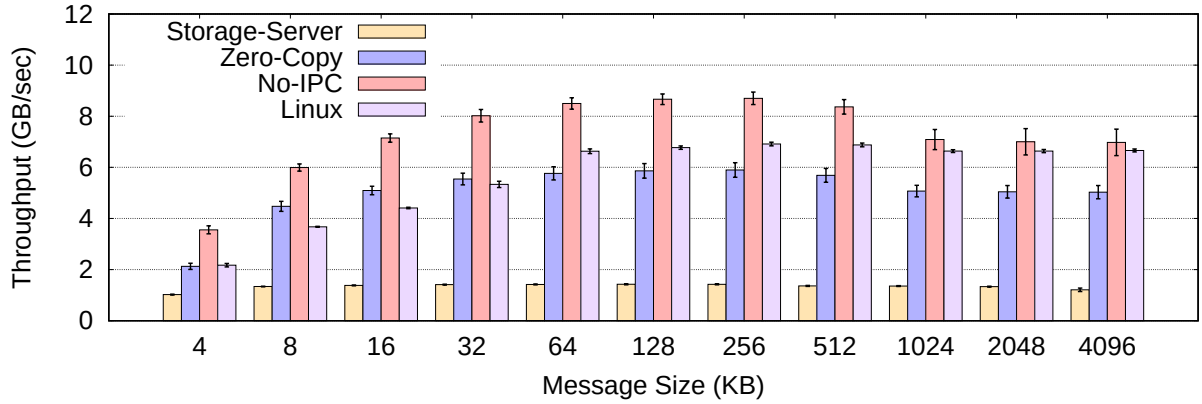


Figure 10.1: Read throughput of microbenchmark-ramdisk.

Figure 10.1 and 10.2 are the throughput of the microbenchmark running with the regular storage server, zero-copy, no-IPC version of it, and Linux. We run the microbenchmark 10 times and calculate a mean and standard deviation. The bars present the mean value of throughput, and the error bars present the standard deviation.

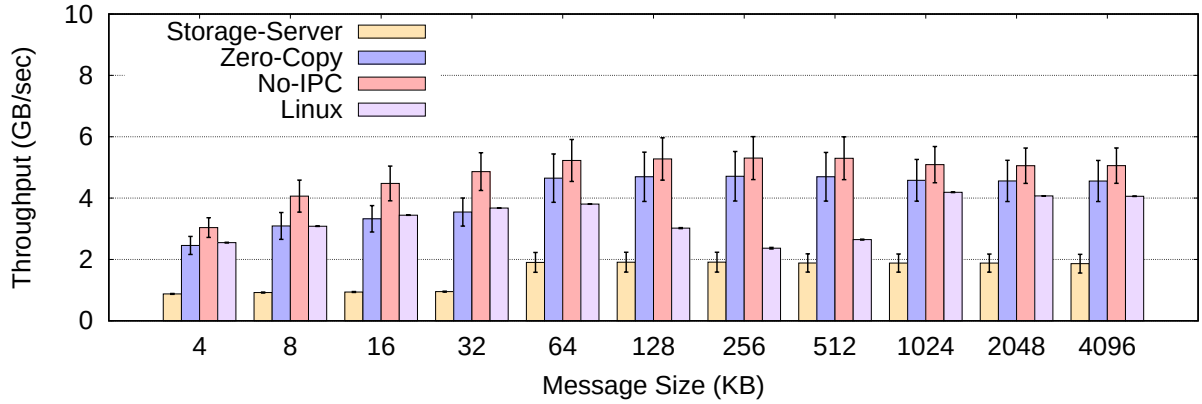


Figure 10.2: Write throughput of microbenchmark-ramdisk.

As shown in the results (see Figure 10.1), the throughput of the storage server with 4KB

message is 1.02GB/sec while the throughput of the zero-copy version is 2.12GB/sec. With other message sizes, the regular storage server's `read` performs `2x` to `4x` slower than the zero-copy version. This is due to the additional copies introduced in the backend and frontend driver. The zero-copy version of the storage server performs faster than Linux until the message size of 32KB. However, Linux outperforms after 64KB because the IPC overhead from the ring buffer increases as the message size increases. On the other hands, the no-IPC version is the fastest in all the message size, because system calls in the monolithic kernel are replaced with normal functions in the rumprun unikernel.

The throughput of `write` (see Figure 10.2) in the storage server is 0.8GB/sec with 4KB messages. However, it bumps up to 1.9GB/sec with 64KB messages. This is because we set the I/O block size to 64KB such that even messages smaller than 64KB are read or written by a 64KB block. In addition, the rump kernel asynchronously batches `write` operations. As a result, all the throughput of the storage server bumps up with the message size larger than 64KB.

10.2.2 NVMe

We evaluate the storage server with an NVMe device and compare it with a rumprun and Linux. The microbenchmark measures the elapsed time for reading/write 4GB of data and calculates the throughput. We set PCI-passthrough for the rumprun unikenel to access the NVMe device directly. Figure 10.3 and 10.4 show the throughput of read and write with different message sizes.

For reading (see Figure 10.3), the storage server achieves half throughput of Linux and the rumprun unikernel. The overhead comes from the ring buffer IPC and additional memory copies. With write operations (see Figure 10.4), the storage server is as fast as the rumprun unikernel. Particularly, message sizes of 2MB and 4MB, the storage server is as fast as Linux.

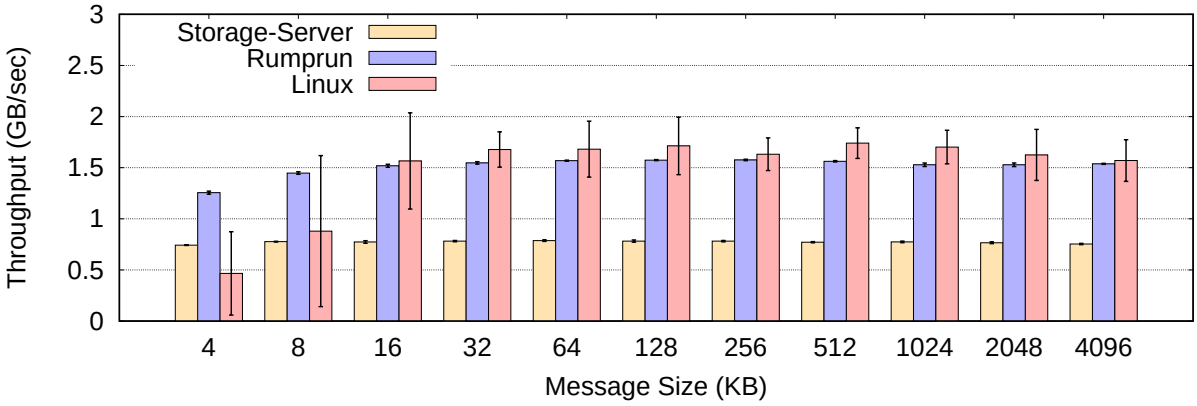


Figure 10.3: Read throughput of microbenchmark-NVMe.

The throughput of the storage and the rumprun unikernel bumps up at the 64KB message size because both set the default I/O block size to 64KB.

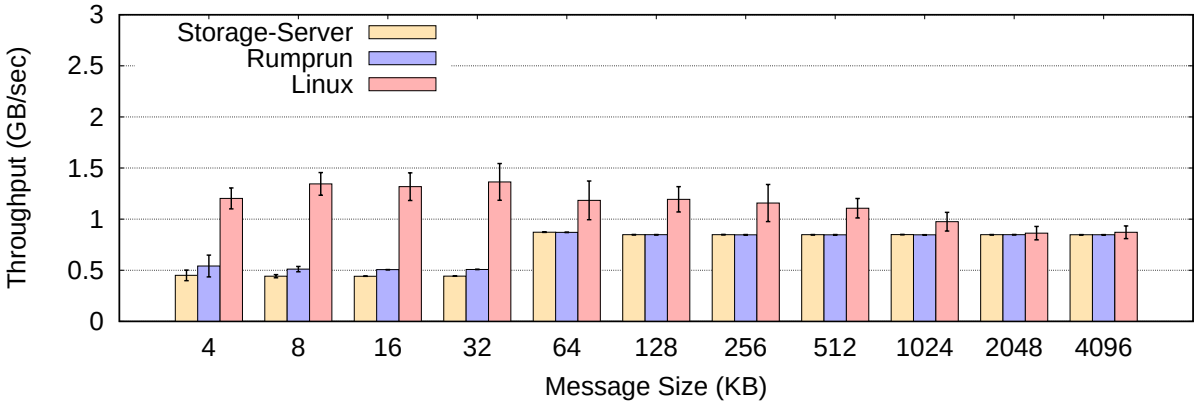


Figure 10.4: Write throughput of microbenchmark-NVMe.

With the large message size, all three are saturated to the same throughput. We attribute it to the NVMe device's performance slow down as it internally does the wear leveling and garbage collecting in case of the write operation.

10.3 Macrobenchmarks

We evaluate the storage server with the macrobenchmarks to see how it works with real applications. A client machine that we use has 10GbE NIC (Intel x520-2 10GbE), 32GB

memory, AMD FX-8350 8-Core Processor, and is connected to the server machine through an InfiniBand [107]. Ubuntu 18.04 with Linux 4.15.0 is running on the client machine.

10.3.1 NFS Server

To evaluate the storage server, we run an NFS erver macrobenchmark by using Sysbench v1.0.11/FileIO [105] from the client machine. We mount an NVMe partition initialized with the ext3 file system and export it through the NFS server.

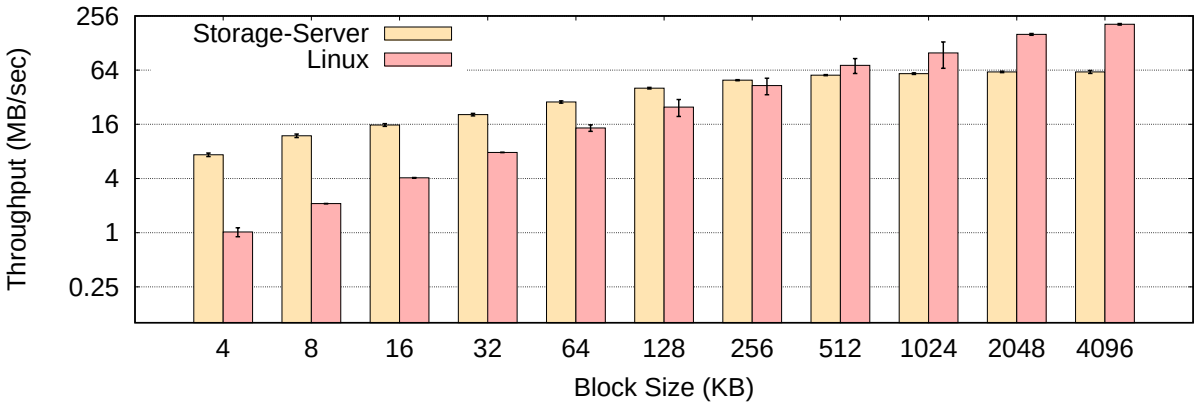


Figure 10.5: Throughput of read of NFS server with Sysbench.

From the client side, we mount the NFS share and run the Sysbench FileIO with different block sizes. We set a single thread, total 30GB of 128 files, 10 seconds for each block size,

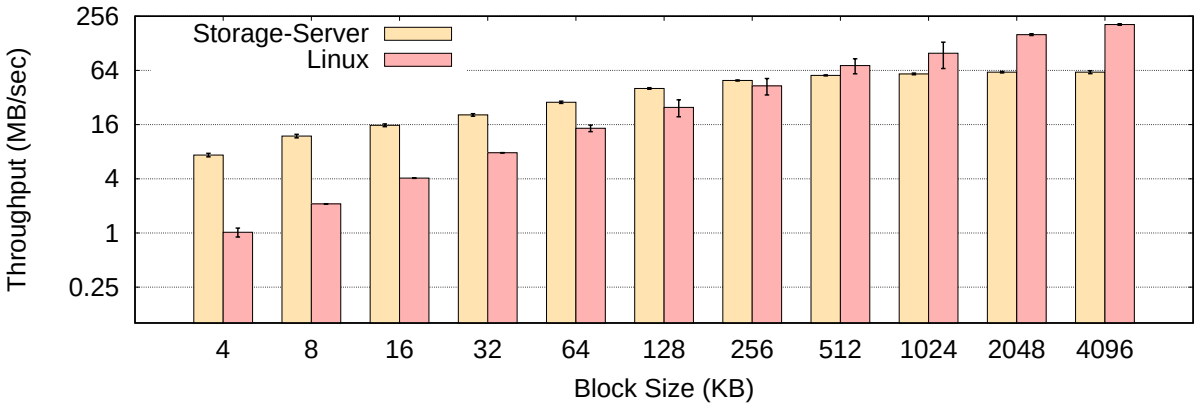


Figure 10.6: Throughput of write of NFS server with Sysbench.

and 1:1 read/write ratio on Sysbench. Also, we set an option ”–file-extra-flags=direct” to avoid page cache on the client side. We ran Sysbench several times and calculated the mean and standard deviation of the throughput of read/write.

Figure 10.5 and 10.6 show the throughput of read and write. For the small block size (less than 512KB), the storage server outperforms Linux. Particularly, the storage server is more than 7 times faster than Linux with 4KB block. This is because the storage server avoids the system call layer which is known to cause a non-negligible performance overhead [100]. However, Linux outperforms the storage server with the large block size. This is because of the additional memory copies and IPC overhead.

Chapter 11

Conclusions and Proposed Post-Preliminary Exam Work

In this dissertation, we presented techniques to provide isolation on software and system components for security and reliability improvement. The intra-unikernel isolation with Intel MPK introduces an isolation scheme for components within a unikernel instance. In particular, it allows us to keep the single address space feature of unikernels and thus maintain their performance benefits by relying on the Intel MPK technology. We demonstrated an overhead as low as 0.6% for macro-benchmarks.

We also introduced the storage server for LibrettOS. Security concerns of the storage system can be mitigated by the strong isolation imposed on the storage server. The unique design of IPC using the lock-free ring buffers allows the storage server to achieve reasonable performance compared with monolithic OSs such as Linux. By building on rumprun unikernel, the storage server can leverage a wide range of NetBSD device drivers without code modification such that the storage server can utilize NVMe storage which is one of the state-of-art hardware. The evaluation demonstrated that the storage server performs reasonably compared with Linux.

Providing isolation of operating systems components has been generally recognized as a performance penalty. For better security, the existing operating systems trade-off the performance. Due to the trade-off, the commodity OSs prioritized better performance and

opted not to have isolation in their components with sacrificing security. From the works in this dissertation, however, we learned that performance and security are not contradicting things, and they are things that can be taken simultaneously.

11.1 Proposed Post-Preliminary Exam Work

We propose four directions for the post-preliminary exam work.

11.1.1 Transparent Fault-Tolerant Storage Server

In this dissertation, we introduce a storage server in LibrettOS. The storage server is built on a rumprun unikernel such that strong isolation is provided by the virtualization. The strong isolation brings security benefits because security vulnerability in the storage system component does not affect the other parts of the system, unlike the monolithic kernel design. Furthermore, the isolation also brings enhanced reliability. Even if the storage system fails for any reason, it can be restored by restarting the storage server. However, since the system has to guarantee that storage I/Os are complete, flying data (data being transmitted) should not be lost.

For storage I/O fault tolerance, storage multipath was introduced and adopted to systems such as Linux and VMWare ESXi virtual machine monitor. The technique of storage multipath defines more than one physical path between computer system components such as CPU and storage devices through buses, controllers, switches, and bridge devices. If one path fails, the system routes the storage I/O to other paths to tolerate the failure. This technique has a downside: frying data can not be restored. Data that has already been transmitted cannot be recovered even if I/O is routed to another path.

Transparent fault tolerance for storage can be achieved by leveraging TCP/IP's fault tolerance [119]. TCP/IP resends the same packets when the acknowledges do not come within a

set of time windows. Like TCP/IP, the frontend driver resends block I/O requests if it does not receive the I/O responses within a time window.

There are several challenges associate with this proposed work. One of them is related to performance. As the frontend driver resends I/O requests, the latency of read and write operations increases and this significantly downgrades the throughput. The time window has to be carefully determined in order that reasonable performance can be achieved. Another challenge is asynchronous storage I/O. The asynchronous storage I/O, such as asynchronous block write returns right after sending the I/O request. There can be a case that an asynchronous write returns even though the I/O request fails.

11.1.2 Transparent Fault-Tolerant File System Server

Since the file system is one of the critical system components in OSs, isolating the file system for security enhancement would be interesting post-preliminary work. Aside from the security benefit, an isolated file system server can bring reliability improvement by supporting fault tolerance. When any fault occurs in the file system, the file system can be recovered by restarting the file system server. Unlike that the fault in the file system requires the entire system to be rebooted, the file system server does not affect the system for recovery.

Unfortunately, data or even metadata could be lost even after the file system recovery because most of them reside in volatile memory. To address the problem, several approaches such as journaling, have been addressed. Furthermore, Chidambaram et al. presented Optimistic Crash Consistency [20] which is a new approach to crash consistency in journaling file systems achieving both a high-level consistency and excellent performance.

These prior works focus on recovery from the file system crash, but none of them are transparent to applications. Therefore, we propose a transparent fault-tolerant file system server for the post-preliminary work. Applications are not aware of any faults that occurred in the

file system server. The file system server should recover from any crash, and not lose any data issued by the applications.

11.1.3 Design and Implementation of Other Servers

Another large part of the system components is device drivers. Device drivers are prone to have security vulnerabilities because they expose large attach surfaces. Therefore, providing isolation to the device drivers can give promising security benefits to systems. Device drivers of USB, sound device, graphic process unit, console device, and so on can be built as device driver servers.

The device driver server can offer potential feasibility of hardware resource disaggregation, or CPU offloading. As a recent trend of hardware devices incorporating more processing power, it enables the system servers to run on the devices' controller and to offload the OS logic from the CPU [95]. Consequently, this provides strong isolation of the device drivers.

For the post-preliminary work, we suggest a device server with enhanced security and reliability, and better performance by offloading execution of device drivers from the CPU.

11.1.4 Fine-grained Intra-Unikernel Isolation

We introduce the technique for intra-unikernel isolation by leveraging Intel MPK feature. We separate the unsafe kernel code from the safe kernel code, and also user code from the kernel code with our isolation technique. Nonetheless, we believe that isolation provision within a user code can be an addition to better security. For example, a formally verified cryptographic library is unlikely to contain vulnerabilities. However, a vulnerability in a user-facing HTTP parsing module (such as CVE2013-2028 [2] in NGINX) can leak the sensitive data that the cryptographic library manipulates (e.g., crpyto keys) when they are in the same user memory address space.

Prior works suggested techniques of intra-process isolation. ELFbac [8] suggested fine-grained intra-process isolation by reusing existing ELF ABI infrastructure. CHERI [113] proposed an extension of conventional processor Intruction-Set Architecutres (ISAs) featuring architectural capabilities to provide fine-grained memory protection and highly scalable software compartmentalization.

We anticipate that these techniques provide fine-grained isolation within application code in unikernels, therefore, propose it as the post-preliminary work.

Bibliography

[1] CVE-2013-1763, 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1763. Online, accessed 5/3/2021.

[2] CVE-2013-2028, 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028. Online, accessed 5/2/2021.

[3] CVE-2014-0226, 2014. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0226. Online, accessed 5/3/2021.

[4] CVE-2016-10229, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10229. Online, accessed 5/3/2021.

[5] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. 1986.

[6] T. E. Anderson. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 92–94, April 1992.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 689–703, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026930.

[8] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. Technical Report TR2013-727, Dartmouth College, Computer Science, Hanover, NH, May 2013. URL http://www.cs.dartmouth.edu/reports/TR2013-727.pdf.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[10] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R Lorch, Barry Bond, Reuben Olinsky, and Galen C Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 239–252. ACM, 2013.

[11] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation*, OSDI'14, 2014.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[14] Kevin Boos and Lin Zhong. Theseus: A State Spill-free Operating System. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 29–35, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5153-9. doi: 10.1145/3144555.3144560. URL http://doi.acm.org/10.1145/3144555.3144560.

[15] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*, CloudCom 2015, pages 250–257. IEEE, 2015.

[16] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[17] T. Bushnell. Towards a new strategy for OS design, 1996. http://www.gnu.org/software/hurd/hurd-paper.html.

[18] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Eld Lazowska. Opal: a single address space system for 64-bit architecture address space. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 80–85. IEEE, 1992.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems.

In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, APSys'11, 2011. ISBN 978-1-4503-1179-3.

[20] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522726. URL https://doi.org/10.1145/2517349.2522726.

[21] Cloudozer LLP. LING/Erlang on Xen website, 2017. http://erlangonxen.org/. Online, accessed 11/20/2017.

[22] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 189–202. ACM, 2011.

[23] Jonathan Corbet. Memory protection keys. *Linux Weekly News*, 2015. https://lwn.net/Articles/643797/.

[24] Intel Corporation. Introduction to Intel® Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

[25] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manual, 2016. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html.

[26] The MITRE Corporation. CVE - Search Results, 2021. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Windows+kernel, Online, accessed 5/3/2021.

[27] Glauber Costa and Don Marti. Redis On OSv. http://blog.osv.io/blog/2014/08/14/redis-memonly/, 2014.

[28] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet'17, pages 36–41. ACM, 2017.

[29] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/cutler.

[30] Martin Decky. HelenOS: Operating System Built of Microservices, 2017. http://www.nic.cz/files/nic/IT_17/Prezentace/Martin_Decky.pdf.

[31] CVE Details. Linux kernel: Vulnerability statistics, 2021. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, Online, accessed 5/3/2021.

[32] Developers. Redox - Your Next(Gen) OS, 2019. https://www.redox-os.org.

[33] Bob Duncan, Andreas Happe, and Alfred Bratterud. Enterprise IoT security and scalability: how unikernels can improve the status Quo. In *IEEE/ACM 9th International Conference on Utility and Cloud Computing*, UUC 2016, pages 292–297. IEEE, 2016.

[34] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th*

ACM Symposium on Operating Systems Principles, SOSP'95, pages 251–266, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917154.

[35] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. HPCVirt'07, 2007.

[36] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill Multiserver Approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 109–114, 2000. URL http://l4ka.org/publications/.

[37] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, 2012.

[38] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 103–116, 2009.

[39] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[40] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 66–77, New York,

NY, USA, 1997. Association for Computing Machinery. ISBN 0897919165. doi: 10.1145/268998.266660. URL https://doi.org/10.1145/268998.266660.

[41] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, 2019.

[42] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing UNIX for Reliability. In *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'06, pages 81–94, 2006. ISBN 3-540-40056-7, 978-3-540-40056-1.

[43] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP Int. Conference*, pages 33–42, 2009.

[44] Jorrit N Herder, David C Van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S Tanenbaum. Dealing with Driver Failures in the Storage Stack. In *Dependable Computing, 2009. LADC'09. 4th Latin-American Symposium on*, pages 119–126, 2009.

[45] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[46] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. Broad New OS Research: Challenges and Opportunities. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HotOS'05, pages 15–15, 2005.

[47] Intel Corporation. Intel® Ethernet 500 Series Network Adapters. https:

//www.intel.com/content/www/us/en/products/network-io/ethernet/ethernet-adapters/ethernet-500-series-network-adapters.html. Online, accessed 5/2/2021.

[48] Antti Kantee. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. In *Ph.D. thesis, Department of Computer Science and Engineering, Aalto University*, 2012.

[49] Antti Kantee and Justin Cormack. Rump Kernels No OS? No Problem! *USENIX; login: magazine*, 2014.

[50] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, page 6, USA, 2003. USENIX Association.

[51] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, page 61, 2014.

[52] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, 2009. ISBN 978-1-60558-752-3.

[53] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[54] Michał Król and Ioannis Psaras. Nfaas: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 134–144. ACM, 2017.

[55] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'17, pages 15–29. ACM, 2017.

[56] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for Multicore Systems: The Sidecore Approach. WIOSCA'07, 2007.

[57] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 49–65, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5.

[58] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 51–57, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5153-9. doi: 10.1145/3144555.3144562. URL http://doi.acm.org/10.1145/3144555.3144562.

[59] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS 2016. ACM, 2016.

[60] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and*

Operating Systems, PLOS'19, pages 8–15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7017-2. doi: 10.1145/3365137.3365395. URL http://doi.acm.org/10.1145/3365137.3365395.

[61] Michael Larabel. The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019. *Phoronix.com*, 2020. URL https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019.

[62] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE journal on selected areas in communications*, 14(7):1280–1297, 1996.

[63] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132786. URL http://doi.acm.org/10.1145/3132747.3132786.

[64] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 175–188, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897916328. doi: 10.1145/168619.168633. URL https://doi.org/10.1145/168619.168633.

[65] Jochen Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[66] Jochen Liedtke. Toward Real Microkernels. *Commun. ACM*, 39(9):70–77, September 1996. ISSN 0001-0782. doi: 10.1145/234215.234473. URL https://doi.org/10.1145/234215.234473.

[67] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[68] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 461–472. ACM, 2013.

[69] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, 2015.

[70] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132763. URL http://doi.acm.org/10.1145/3132747.3132763.

[71] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX

Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616491.

[72] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 399–413, 2019. ISBN 978-1-4503-6873-5.

[73] NetBSD Contributors. The NetBSD Project, 2019. http://netbsd.org/.

[74] NetBSD Contributors. The NetBSD Project: Announcing NetBSD 9.0, 2020. https://www.netbsd.org/releases/formal-9/NetBSD-9.0.html.

[75] NetBSD Contributors. nvme(4) - NetBSD Manual Pages, 2021. http://man.netbsd.org/NetBSD-8.0/i386/nvme.4.

[76] Newlib. Newlib website, 2017. https://sourceware.org/newlib/. Online, accessed 12/12/2017.

[77] Ruslan Nikolaev. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-126-9. doi: 10.4230/LIPIcs.DISC.2019.28. URL http://drops.dagstuhl.de/opus/volltexte/2019/11335.

[78] Ruslan Nikolaev and Godmar Back. VirtuOS: An Operating System with Kernel

Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 116–132, 2013. ISBN 978-1-4503-2388-8.

[79] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 114–128, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi: 10.1145/3381052.3381316. URL https://doi.org/10.1145/3381052.3381316.

[80] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 1–14, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050758. URL http://doi.acm.org/10.1145/3050748.3050758.

[81] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'19, 2019.

[82] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC19)*, pages 241–254, 2019.

[83] PCI-SIG. Single Root I/O Virtualization and Sharing Specification, 2019. http://pcisig.com/specifications/iov/.

[84] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System

is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, 2014. ISBN 978-1-931971-16-4.

[85] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950399. URL http://doi.acm.org/10.1145/1950365.1950399.

[86] The Rust Project. Global allocators - The Edition Guide, 2019. https://doc.rust-lang.org/edition-guide/rust-2018/platform-and-target-support/global-allocators.html.

[87] The Rust Project. Application Binary Interface - The Rust Reference, 2019. https://doc.rust-lang.org/reference/abi.html#the-link_section-attribute.

[88] The Rust Project. Macros - The Rust Programming Language, 2019. https://doc.rust-lang.org/1.29.0/book/first-edition/macros.html.

[89] The Rust Project. unsafe Rust - The Rust Programming Language, 2019. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[90] The Rust Project. Rust Programming Language, 2019. https://www.rust-lang.org.

[91] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[92] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 54, 2010.

[93] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A Framework for Building per-Application Library Operating Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 671–688, USA, 2016. USENIX Association. ISBN 9781931971331.

[94] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization*, IISWC 2011, pages 137–148. IEEE, 2011.

[95] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/shan.

[96] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, and Yubin Xia. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.

[97] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, 2019.

[98] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *The Network and Distributed System Security Symposium*, NDSS'17, 2017.

[99] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot topics in Middleboxes and Network Function Virtualization*, HotMiddlebox 2016, pages 44–49. ACM, 2016.

[100] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924946.

[101] SPDK Contributors. Storage Performance Development Kit (SPDK), 2019. http://spdk.io/.

[102] SSRG-VT. Rumprun-smp, 2020. https://github.com/ssrg-vt/rumprun-smp/tree/c166f324f57456dedc7b79711f16ed09fe0e71b4, Online, accessed 04/18/2021.

[103] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[104] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 119–130, 1995.

[105] Sysbench Contributors. SysBench 1.0: A System Performance Benchmark, 2019. http://github.com/akopytov/sysbench/.

[106] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292. USENIX Association, 2006.

[107] Mellanox Technologies. Introduction to InfiniBand™. URL https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[108] The Linux Foundation. Data Plane Development Kit (DPDK), 2019. http://dpdk.org/.

[109] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, page 9. ACM, 2014.

[110] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference*, ATC 2017, page 8, 2017.

[111] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. *USENIX Security Symposium*, 2019.

[112] Alexander Warg and Adam Lackorzynski. L4Re Runtime Environment, 2018. http://l4re.org/.

[113] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical report, University of Cambridge, Computer Laboratory, 2019. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[114] Carsten Weinhold and Hermann Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 81–93. ACM, 2008.

[115] Adam Wick. The HaLVM: A Simple Platform for Simple Platforms. Xen Summit, 2012.

[116] Xen Project. Understanding the Virtualization Spectrum, 2014. https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum.

[117] Xen Project. Paravirtualization (PV), 2015. https://wiki.xenproject.org/wiki/Paravirtualization_(PV).

[118] Xen Website. Google Summer of Code Project, TinyVMI: Porting LibVMI to Mini-OS, 2018. https://blog.xenproject.org/2018/09/05/tinyvmi-porting-libvmi-to-mini-os-on-xen-project-hypervisor/, Online, accessed 10/30/2018.

[119] L Zhang. Why TCP Timers Don't Work Well. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures & Protocols*, SIGCOMM '86, page 397–405, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897912012. doi: 10.1145/18172.18216. URL https://doi.org/10.1145/18172.18216.

[120] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A Dynamic Library

Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018.

[121] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.