On Improving the Security of Virtualized Systems through Unikernelized Driver Domain and Virtual Machine Monitor Compartmentalization and Specialization

A K M Fazla Mehrab

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Binoy Ravindran, Chair

Daphne Yao

Paul Plassmann

Haibo Zeng

Ruslan Nikolaev

February 17, 2023

Blacksburg, Virginia

Keywords: Operating Systems, Unikernels, Virtualization, Hypervisors, VMM, Compartmentalization

On Improving the Security of Virtualized Systems through Unikernelized Driver Domain and Virtual Machine Monitor Compartmentalization and Specialization

A K M Fazla Mehrab

(ABSTRACT)

Virtualization is the backbone of cloud infrastructures. Its core subsystems include hypervisors and virtual machine monitors (VMMs). They ensure the isolation and security of co-existent virtual machines (VMs) running on the same physical machine. Traditionally, driver domains – isolated VMs in a hypervisor such as Xen that run device drivers – use general-purpose full-featured OSs (e.g., Linux), which has a large attack surface, evident by the increasing number of their common vulnerabilities and exposures (CVEs). We argue for using the unikernel operating system (OS) model for driver domains. In this model, a single application is statically compiled together with the minimum necessary kernel code and libraries to produce a single address-space image, reducing code size by as much as one order of magnitude, which yields security benefits.

We develop a driver domain OS, called *Kite*, using NetBSD OS's rumprun unikernel. Since rumprun is directly based on NetBSD's code, it allows us to leverage NetBSD's large collection of device drivers, including highly specialized ones such as Amazon ENA. Kite's design overcomes several significant challenges including Xen's limited para-virtualization (PV) I/O support in rumprun, lack of Xen backend drivers which prevents rumprun from being used as a driver domain OS, and NetBSD's lack of support for running driver domains in Xen. We instantiate Kite for the two most widely used I/O devices, storage and network, by designing

and implementing the storage backend and network backend drivers. Our evaluations reveal that Kite achieves competitive performance to a Linux-based driver domain while using 10x fewer system calls, mitigates a set of CVEs, and retains all the benefits of unikernels including a reduced number of return-oriented programming (ROP) gadgets and advanced gadget-related metrics.

General-purpose VMMs include a large number of components that may not be used in many VM configurations, resulting in a large attack surface. In addition, they lack intra-VMM isolation, which degrades security: vulnerabilities in one VMM component can be exploited to compromise other components or that of the host OS and other VMs (by privilege escalation). To mitigate these security challenges, we develop principles for VMM compartmentalization and specialization. We construct a prototype, called *Redwood*, embodying those principles. Redwood is built by extending Cloud Hypervisor and compartmentalizes thirteen critical components (i.e., virtual I/O devices) using Intel MPK, a hardware primitive available in Intel CPUs. Redwood has fifteen fine-grained modules, each representing a single feature, which increases its configurability and flexibility. Our evaluations reveal that Redwood is as performant as the baseline Cloud Hypervisor, has a 50% smaller VMM image size and 50% fewer ROP gadgets, and is resilient to an array of CVEs.

I/O acceleration architectures, such as Data Plane Development Kit (DPDK) enhance VM performance by moving the data plane from the VMM to a separate userspace application. Since the VMM must share its VMs' sensitive information with accelerated applications, it can potentially degrade security. The dissertation's final contribution is the compartmentalization of a VM's sensitive data within an accelerated application using the Intel MPK hardware primitive. Our evaluations reveal that the technique does not cause any degradation in I/O performance and mitigates potential attacks and a class of CVEs.

On Improving the Security of Virtualized Systems through Unikernelized Driver Domain and Virtual Machine Monitor Compartmentalization and Specialization

A K M Fazla Mehrab

(GENERAL AUDIENCE ABSTRACT)

Instead of using software on a local device like a laptop or a mobile phone, consumers can access the same services from a remote high-end computer through high-speed Internet. This paradigm shift in computing is enabled by a remote computing infrastructure known as the "cloud," wherein networked server computers are deployed to execute third-party applications, often untrusted. Multiple applications are consolidated on the same server to save computer resources, but this can compromise security: a malicious application can steal co-existent applications' sensitive data. To enable resource consolidation and mitigate security attacks, applications are executed using a virtual machine (VM) – an abstract machine that runs its own operating system (OS). Multiple VMs run on a single physical machine using two software systems: hypervisor and virtual machine monitor (VMM). They ensure that VMs are spatially isolated from each other, localizing security attacks. This dissertation focuses on enhancing the security of hypervisors and VMMs.

The hypervisor and VMM have multiple responsibilities toward supporting the OS running on the physical computer and VMs. The OS runs software called device drivers, which communicate with input-output (I/O) hardware such as network and storage devices. Device drivers, usually written by third-party and I/O device manufacturers, are highly vulnerable to security attacks. To mitigate such attacks, device drivers are often run inside special VMs,

called driver domains. State-of-the-art driver domains use a general-purpose full-featured OS such as Linux, which has a large code base (in the tens of millions of lines of code) and thus, a large attack surface. To address this security challenge, the dissertation proposes using lightweight, single-purpose VMs called unikernels, as driver domain OSs. Their code size is smaller than that of full-featured OSs by as much as one order of magnitude, which yields security benefits.

We design and develop a unikernel-based driver domain, called *Kite*, for network and storage I/O devices. Kite uses NetBSD OS's rumprun unikernel for creating a driver domain OS. Using rumprun unikernel as a driver domain OS requires overcoming many technical challenges including a lack of support in a popular hypervisor such as Xen for performing I/O operations and communicating with rumprun, among others. Kite's design overcomes these challenges. Our empirical studies reveal that Kite is ten times less likely to be affected by future attacks and ten times faster to start than existing solutions for driver domains. At the same time, Kite domains match the performance of state-of-the-art driver domain OSs such as Linux.

The hypervisor and VMM are responsible for creating VMs and providing resources such as memory, processing power, and hardware device access. Existing VMMs are designed to be versatile. Thus, they include a large number of components that may not be used in many VM configurations, resulting in a large attack surface. In addition, VMM components are not well spatially separated from each other. Thus, vulnerabilities in one component can be exploited to compromise other components. To address these security challenges, the dissertation proposes a set of principles for i) customizing a VMM for each VM's needs, instead of using one VMM for all VMs, and ii) strongly isolating VMM components from each other. We realize these principles in a prototype implementation called *Redwood*. Red-

wood is highly configurable and separates critical I/O components from each other using a hardware primitive. Our evaluations reveal that Redwood significantly reduces the VMM's size and VMM's vulnerabilities while maintaining performance.

To enhance VM performance, I/O acceleration software is often used that eliminates communication overheads in the VMM. To do so, the VMM must share VMs' sensitive information with accelerated applications, which can potentially degrade security. The dissertation's final contribution is a technique that strongly isolates and limits access to sensitive information in the application using a hardware primitive. Our evaluations reveal that the technique improves security by localizing attacks without sacrificing performance.

Dedication

To my late father, Md. Insan Ali.

Acknowledgments

Working with Professor Binoy Ravindran has been a tremendous honor and privilege. I am deeply grateful for this opportunity and his guidance, continued support, and invaluable advice. His professional demeanor, strong principles, and warm personality inspire me greatly. I will always treasure the time I spent learning from him and strive to emulate his high standards for my future endeavor.

Professor Ruslan Nikolaev's wisdom and expertise have greatly enriched my knowledge and helped shape my academic path. I am grateful for his precious guidance and help throughout my PhD journey. I will cherish the time I spent working with him.

I am sincerely thankful to my committee members, Professor Daphne Yao, Professor Paul Plassmann, and Professor Haibo Zeng, for taking time out of their busy schedules to review my dissertation and offer valuable feedback.

I am deeply grateful for the support and guidance I received from the talented individuals at SSRG, past and present. While it is difficult to name everyone, I especially want to recognize Professor Pierre Olivier, a close mentor since my master's studies. His kind support, the teaching of research methodology, and advice inspired me to continue this journey.

Our band (The Migratory Birds of Blacksburg) and the Bangladeshi community in this beautiful town have relieved my stress and provided unforgettable memories. Their presence made my PhD journey much easier, and I am truly thankful for their support.

My gratitude extends to my parents, brothers, and other family members for their sacrifices and support. Finally, I would like to express my appreciation to my loving wife, Swarna, for being my constant companion throughout my journey, providing continued support, her sacrifice, and delicious dishes, which made my life much easier.

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

This research is also based upon work supported by the U.S. Office of Naval Research (ONR) under grants N00014-18-1-2022, N00014-19-1-2493, N00014-16-1-2104, and N00014-16-1-2711.

Contents

List of Figures xviii

List of Tables xxiv

Attribution xxvi

1 Introduction 1

1.1 The Unikernel Operating System Model 5

1.2 VMM's Security Threats in Clouds 8

1.3 I/O Acceleration for VMs 10

1.4 Summary of Research Contributions 11

1.4.1 Kite: Unikernelized Storage Domain 11

1.4.2 Kite: Unikernelized Network Domain 12

1.4.3 Redwood: Flexible Secure VMM 13

1.4.4 Vhost User Compartmentalization in DPDK 13

1.5 Dissertation Organization 14

2 Background 16

2.1 Xen 16

2.2 KVM 17

2.3 Xen I/O Drivers 17

2.4 Xen Blkfront and Blkback 18

2.5 Xen Netfront and Netback 19

2.6 Xen Driver Domain 20

2.7 Virtual Machine Monitor 21

2.8 Virtio PV Devices 23

2.9 OVS-DPDK Vhost User 24

2.10 Unikernel 26

2.11 Rump Kernels and Rumprun 26

2.12 Memory-based Isolation 28

3 Related Work **30**

3.1 Hypervisor Disaggregation Approaches 30

3.2 Unikernels for Cloud Infrastructures 33

3.3 Driver Domain and Backend Improvement 34

3.4 Protection Key-based Memory Isolation 36

3.5 Virtual Machine Monitors 37

4 Kite: Unikernelized Storage Domain **40**

4.1 Challenges 40

4.2 Threat Model 41

4.3 Storage Device Driver 43

4.4 Storage Backend Driver 44

4.5 Storage Domain Application 49

5 Kite's Storage Domain Prototype **50**

5.1 Block Device Interface 50

5.2 Blkback Instantiation 53

5.3 Blkback Initialization and Connection 55

5.4 Event Handler and Request Handler Thread 55

5.5 Handling Device Driver Responses 56

5.6 Persistent Reference 57

5.7 Indirect Segments 58

5.8 Application 58

5.9 Implementation Effort 59

6 Kite: Unikernelized Network Domain **60**

6.1 Challenges 60

6.2 Threat Model 62

6.3 Network Device Driver and Interface 63

6.4 Netback Driver 63

6.5 Linking Netback With a Physical Device 66

7 Kite's Network Domain Prototype **68**

7.1 Virtual Network Interface 68

7.2 Netback Instantiation and Connection 69

7.3 Transmit 71

7.4 Receive 72

7.5 Threaded Implementation 73

7.6 Physical Network Device Driver 74

7.7 Bridging Application 75

7.8 Implementation Effort 76

8 Kite's Storage Domain Evaluation **77**

8.1 Experimental Setup 77

8.2 dd 79

8.3 SysBench 80

8.3.1 SysBench File I/O 80

8.3.2 SysBench MySQL 81

8.4 Filebench 82

8.4.1 Filebench File server 82

8.4.2 Filebench MongoDB Server 83

8.4.3 Filebench Web server 84

9 Kite's Network Domain Evaluation **85**

9.1 Experimental Setup 85

9.2 Nuttcp 87

9.3 Latency 88

9.4 Apache 88

9.5 Redis 90

9.6 MySQL 90

10 Kite's Security Evaluation **92**

10.1 Image Size and Boot Time 92

10.2 ROP Gadget 93

10.3 Gadget Set Analysis 95

10.4 Syscall Reduction and CVEs 96

11 Redwood: Flexible Secure VMM **99**

11.1 Design Principles 99

11.2 Challenges 101

11.3 Trusted Computing Base and Trust Model 102

11.4 Achieving Per-VM Specialization 103

11.4.1 VM Bootloading 105

11.4.2 Virtual I/O Device 106

11.4.3 VM Migration 108

11.5 Establishing Intra-VMM Isolation 109

11.5.1 What to Compartmentalize? 110

11.5.2 How to Compartmentalize? 110

11.5.3 When to Enable/Disable a Compartment? 111

11.6 Securing OVS-DPDK Vhost User 111

12 Implementation of Redwood **114**

12.1 Workload-aware Redwood 114

12.2 Isolation inside Redwood 116

12.3 Unikernel Support 119

12.4 Isolating Vhost User in OVS-DPDK 119

12.5 Implementation Effort 120

13 Redwood's Performance Evaluation **122**

13.1 Experimental Setup for Performance Evaluation 122

13.2 Virtio Network Device Performance 123

13.2.1 iPerf 123

13.2.2 Apache 124

13.2.3 Network Latency 125

13.3 Virtio Block Device Performance 126

13.3.1 dd 127

13.3.2 SysBench File I/O 127

13.4 Virtio Balloon Performance 128

13.5 Vhost User Network Device Performance 129

13.5.1 iPerf 130

13.5.2 Redis 130

14 Redwood's Security Evaluation **132**

14.1 CVE Analysis 132

14.2 Image Size Reduction 135

14.3 ROP Gadget Reduction 136

14.4 Gadget Set Analysis 136

15 Conclusions **139**

15.1 Contributions Revisited 140

15.1.1 Kite: Unikernelized Storage Domain 140

15.1.2 Kite: Unikernelized Network Domain 141

15.1.3 Redwood: Flexible Secure VMM 141

15.1.4 Vhost User Compartmentalization in DPDK 142

15.2 Perspective on Dissertation Contributions 142

15.3 Limitations and Future Work 144

15.3.1 Ensuring Protection Key's Integrity in Redwood 144

15.3.2 Automating Compartmentalization 145

15.3.3 Unikernelized Vhost User Devices 146

15.3.4 Unikernelized Virtio Driver Domain for Xen and KVM 147

15.3.5 Rumprun PVH 148

Bibliography **151**

List of Figures

1.1 CVEs for drivers during 2001–2019 [41]. 2

1.2 Category of QEMU CVEs [42]. 4

1.3 Traditional VM model versus the unikernel VM model. 6

1.4 Kite and Ubuntu network domain: image size, boot time, and syscall comparison. 6

1.5 ROP gadget comparison. For a detailed gadget breakdown of rumprun/Kite and advanced gadget metrics, see Chapters 10 and 14. 8

2.1 Xen hypervisor runs on bare metal hardware, and guest VMs run directly on the hypervisor layer, where the administrative OS also runs inside a guest VM. Xen has its implementation for resource distribution and VM management. KVM module converts a Linux (Host) into a hypervisor, where guest VMs are managed with the help of VMM inside the host OS that distributes resources between VMs. 18

2.2 Xen's PV I/O device driver model. 20

2.3 Different modes of virtio devices. 23

2.4 OVS-DPDK architecture. 25

2.5 Rumprun stack on Xen. 27

4.1 Xen's PV storage driver model. 42

4.2 Rumprun storage domain. 43

4.3 Blkfront and blkback communication. For read operations, blkfront places a request in the ring and notifies blkback. Upon receiving the notification, blkback reads data from the storage device, places it in the shared memory buffer, and sends a notification to blkfront so that it can read from the buffer. For write operations, blkfront places the data in the shared memory, places a write request in the ring, and sends a notification to blkback. Upon receiving the notification, blkback writes the data in the storage device and sends a notification to blkfront upon completion. 45

4.4 Batching multiple segments improves storage domain performance. However, there is no guarantee that segments in a disk (A) will be accessed consecutively. Typically, it would take 6 separate read/write operations to access the segments marked with blue boxes (B). However, using batching, we can perform one operation for segments 2, 3, and 4 combined and another for segments 5 and 6 combined, resulting in a total of 3 operations instead of 6, as shown with the green boxes (C). In this scenario, batching saves half the time required to perform all operations separately. 47

4.5 Both direct **(a)** and indirect **(b)** requests can point to one 4K page per reference, with a maximum of 11 and 8 references per request, respectively. A page referenced by a direct request contains segment data. In contrast, a page referenced by an indirect request can contain up to 512 other references, each pointing to a 4K page containing segment data. 48

5.1 Xenstore is a key-value store database that maintains a directory-like structure for keys, each representing a path. This listing is a partial snapshot of Xenstsore database generated using the `xenstore-ls` command. It shows some key-value pairs for the storage driver domain (ID 2), which is connected to a blkfront from guest domain (ID 4). 53

6.1 Xen's PV network driver model. 61

6.2 Rumprun network driver domain architecture. 62

6.3 Netfront and netback communication. To send packets from DomU, netfront places packets in the shared memory, a transmit request in the Tx ring, and notifies netback. Being notified, netback copies those packets, forwards them to the NIC through netback's *vif*, and notifies netfront upon completing forwarding. A similar sequence of operations is performed using the Rx ring for forwarding network packets from a *vif* to DomU. 64

6.4 Linking netbacks can help multiplex a single NIC between multiple guest VMs. Linking methods, like network bridges, can be utilized reduce the need for multiple NICs. 66

7.1 Data exchange between netback and netfront. 72

7.2 Threaded implementation of netback for efficient interaction with netfront. . 73

8.1 Throughput measurements using dd. 79

8.2 File I/O Throughput measurements using sysbench. 80

8.3 MySQL throughput measurements using SysBench. 81

8.4 Fileserver throughput measurements using filebench. 82

8.5 MogoDB server performance measurement using filebench. 83

8.6 Web server performance measurement using filebench. 84

9.1 Nuttcp throughput for UDP file transfers. 87

9.2 Latency comparison for Linux and rumprun network driver domains. 88

9.3 Apache throughput varying file size. 89

9.4 Throughput, transfer time, and request rate. 89

9.5 Redis key-value store throughput. 90

9.6 MySQL throughput. 91

9.7 CPU utilization while benchmarking the MySQL. 91

10.1 Image size and boot time comparison. 93

10.2 ROP gadget comparison. 94

10.3 Gadget set analysis. 95

10.4 System call count comparison. 97

11.1 An example of a virtualization stack focusing on components for VMs using PV storage. The host OS, such as Linux, manages the physical storage device (e.g. NVMe SSD) driver, a filesystem (e.g., Ext-4), a virtual filesystem providing a common interface, and a hypervisor (e.g., KVM). The virtual disk images are files of specific formats (e.g., raw and qcow) in the host OS. VMM (e.g., QEMU) instances provide implementations for these formats and PV block devices (similar to blkback in Xen). A guest OS runs the PV block driver and filesystems in its kernel space for offering storage interfaces to applications. 107

11.2 QEMU CVE trend from 2007 to 2022. [41] 108

11.3 The OVS-DPDK architecture provides faster networking for guest VMs by running as a userspace process on the host and utilizing vhost-user interfaces and an OVS switch. It connects to physical NICs through the NIC driver in the kernel or the PMD driver in DPDK and establishes the control path with the VMM and data paths with the guest VMs through virtio-net or virtio-pmd drivers. However, it also poses a security risk as the OVS-DPDK process has unfiltered access to all vhost-user instances, which may contain sensitive information from the guest VMs. 112

12.1 Core and additional components of Redwood. 115

13.1 iPerf3 TCP throughputs for receive and transmit over network. 124

13.2 Apache server throughput measured using Apache benchmark. 125

13.3 Network Latency measured using different tools. 126

13.4 Virtio block's read and write throughput measured using dd. 127

13.5 File I/O throughput measured using SysBench. 128

13.6 Memory bandwidth while using ballooning. 129

13.7 TCP receive and transmit throughput for vhost-user-net devices. 130

13.8 Redis throughput for operations under varying degrees of request concurrency. 131

14.1 Image sizes for Cloud Hypervisor (All) and different configurations of Redwood. 136

14.2 Gadgets for Cloud Hypervisor (All) and different configurations of Redwood. 137

14.3 Gadget set analysis. 137

15.1 Unikernelized acceleration architectures with virtio-vhost-user devices can provide VM-level isolation to reduce the attack surface. 147

15.2 Architecture for unikernelized virtio driver domain on Xen. 148

15.3 Overview of the various virtualization modes implemented in Xen. tesy: Xen Project [29]) 149

List of Tables

3.1 Prior arts related to Kite's approach. 31

3.2 Prior arts related to Redwood's approach. 38

5.1 Key functions in Kite storage domain. 51

5.2 Important storage device information available in Xenstore. 54

5.3 Lines of code (LOC) changed or added for Kite storage domain. 59

7.1 Important network device information available in Xenstore. 69

7.2 Key functions in Kite network domain. 75

7.3 LOC changed or added for Kite network domain. 76

8.2 Configuration of Xen domains on the server side. 78

8.1 Hardware configuration. 78

9.1 Hardware configuration. 86

9.2 Configuration of Xen domains on the server side. 86

10.1 Examples of CVEs prevented by only keeping necessary system calls. 97

11.1 Example of emulated and paravirtualized devices in VMMs. 106

12.1 Isolated virtio PV devices in Redwood. 117

12.2 LOC changed or added for Redwood prototype. 121

13.1 Hardware configuration. 123

Attribution

The contents of this dissertation are based on my own work, which has already been documented in three papers. The dissertation's principal contributions have been documented in two papers; one of them is published (ACM Eurosys'22), and the other is under review at the time of writing the dissertation. My third paper (ACM HPDC'19) focuses on unikernels in the HPC domain. Although it is not part of the dissertation, it was the foundation and inspiration for my dissertation's work on unikernels.

- A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. 2023. Rethinking Virtual Machine Monitors for Better Security in Clouds. Conference paper submitted in January 2023, currently under review.

- A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. 2022. Kite: Lightweight Critical Service Domains. *In Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 384–401. https://doi.org/10.1145/3492321.3519586

- Pierre Olivier, A K M Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Rob Lyerly, and Binoy Ravindran. 2019. HEXO: Offloading HPC ComputeIntensive Workloads on Low-Cost, Low-Power Embedded Systems. *In The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 85-96. https://doi.org/10.1145/3307681.3325408

Chapter 1

Introduction

Cloud infrastructures run thousands of server machines, each of which facilitates a multi-tenant environment so that different applications with varying resource requirements can execute seamlessly and transparently. Such an environment should isolate tenants from one another in order to ensure data and execution integrity. A separate virtual machine (VM), equipped with the necessary resources and capable of running the preferred operating system (OS), is often allocated per tenant so that each tenant's workload can be executed securely and (usually) with no modifications. The ability to execute many VMs simultaneously on the same server, with strong isolation between them and the relative ease of managing their resources has made virtualization technology the backbone of modern cloud infrastructures.

Hypervisors and virtual machine monitors (VMMs) are essential components that enable virtualization. They are often described interchangeably due to their overlapping and interdependent responsibilities. In this dissertation, we use the term *hypervisor* to refer to software that performs core operations such as scheduling and resource allocation while running on bare metal (e.g., XEN [49], VMware ESX [104]) or that utilizes a host OS (e.g., KVM [76], VirtualBox [135]) for virtualization. We use the term *VMM* to refer to software that performs other essential virtualization operations with the help of a hypervisor. Examples include QEMU [66], Firecracker [118], and Cloud Hypervisor [45].

The role of VMMs varies depending on the hypervisor being used, due to differences in architecture among hypervisors. For example, QEMU with KVM can create a VM, allocate

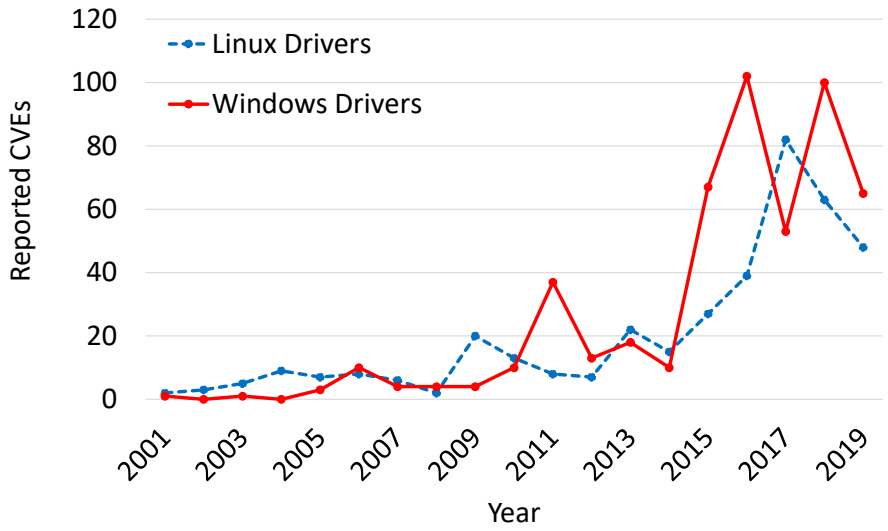


Figure 1.1: CVEs for drivers during 2001–2019 [41].

memory and virtual CPUs (vCPUs), help boot a guest OS, emulate virtual I/O devices, migrate VMs between servers, and stop VMs. Therefore, VMMs have a significant role in ensuring functionality, flexibility, performance, security, and reliability in any virtualization environment. In contrast, QEMU is only required for device emulation when using Xen in the hardware virtual machine (HVM) mode. Although hypervisors and VMMs are critical for VM-level isolation, they contribute to the size of the trusted computing base (TCB).

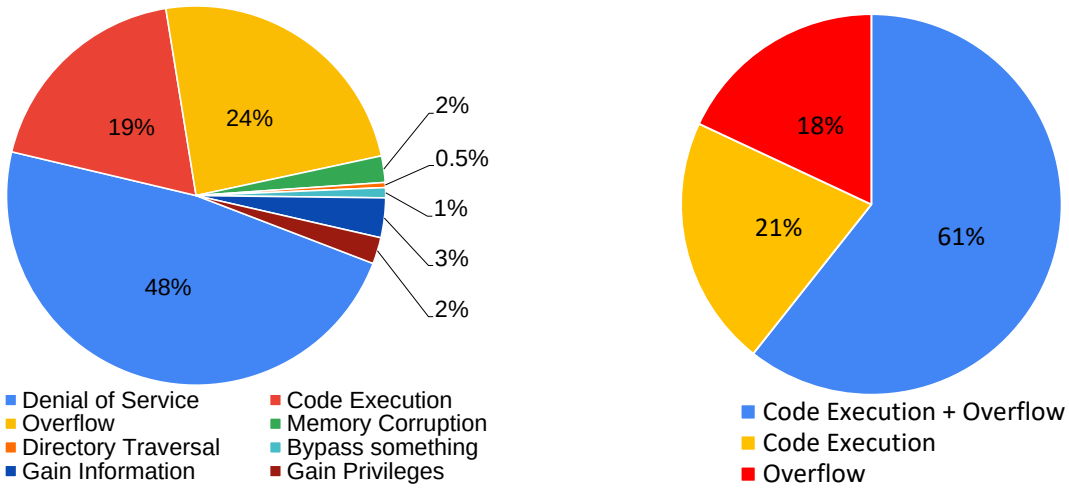
Virtualization ecosystems must provide efficient and secure access to network and storage I/O devices. To that end, hypervisors such as Xen [49], which is used by Amazon EC2 [39] and Rackspace cloud servers [43], have developed a number of solutions over the years. First, network and storage I/O are exposed to guest OSs via efficient para-virtualized (PV) drivers which communicate with the corresponding physical device drivers. Second, a physical device driver (e.g., a network or storage driver) can be executed in a separate *driver domain* [140] for more effective load balancing and enhanced security. Isolating drivers in separate VMs is especially important as the number of common vulnerabilities and exposures (CVEs) for drivers continue to surge across different OSs (see Figure 1.1).

Driver domains are special guest OSs, which run inside isolated VMs. They are often advocated for enhanced isolation and security [114]: they can be used to house certain classes of device drivers. Thus, instead of running device drivers in privileged *service VMs* such as Xen's administrative OS, called Dom0 [6], where a potential driver vulnerability can be catastrophic for the entire system, they are isolated in driver domains. By removing the drivers from Dom0, Dom0's attack surface is also reduced, further increasing security.

A downside of service VMs is that they are relatively heavy-weight as they usually run a general-purpose, full-featured OS such as Linux. Such VMs are cumbersome for deployment and upgrades. Linux is designed to support many subsystems (e.g., audio, video, USB) as well as user-space libraries, tools, daemons, and configuration scripts that are irrelevant to network or storage drivers alone, and yet, they all need to be properly maintained when Linux runs as a driver domain. Case in point: Ubuntu Server 18.04's image is ≈1GB; the kernel alone, without any modules, is ≈50MB [11].

General-purpose OSs are also not ideal for security as they expose a potentially large attack surface, which is undesirable for systems with a greater degree of resource sharing such as Amazon EC2. Even special stripped-down distributions, though rarely used in practice, still have large memory footprints, which add up in enterprise-scale bare-metal cloud systems that handle many I/O devices. Moreover, it is simply impossible to reduce the number of system calls since many of them (e.g., clone, exec, file I/O, etc) are essential for running a Linux-based OS. In Chapter 10, we show that the number of required Linux system calls for running a driver domain is as high as 171.

On the other hand, VMM instances contribute to the TCB size as they run as userspace processes in the host/administrative OS. VMM bugs are a significant concern due to their potential to compromise the main security benefit of virtualization. Many vulnerabilities related to VMMs are reported year-round, with QEMU being a particularly high-profile



(a) QEMU CVEs from 2007 to 2022. (b) CVEs with a minimum score of 6.5.

Figure 1.2: Category of QEMU CVEs [42].

example. In fact, we analyzed and categorized 443 QEMU CVEs reported from 2007 to 2022; Figure 1.2a shows the results. These vulnerabilities, which include denial of service (DoS), memory overflow, and arbitrary code execution attacks represent a variety of security flaws that require different security measures for mitigation.

Many of these bugs arise from improper memory handlings, such as null pointer dereferencing, use-after-free, and out-of-bound access, which are common in software written in C, a language that lacks many safety features. Rust, a language that features type systems and supports ownership models can help prevent many of these vulnerabilities. However, no language can guarantee complete safety because human error can occur during software development. For example, hundreds of CVEs have been reported for Rust, despite it being a relatively newer language. Furthermore, memory safety techniques are often not sufficient to mitigate all vulnerabilities. For instance, an incomplete validation in the Rust-based Firecracker VMM led to CVE-2019-18960 [12], which could potentially allow attackers to perform memory manipulation or remote code execution exploits.

Figure 1.2b categorizes 61 high-impact QEMU CVEs with a minimum score[1] of 6.5. Only 18% of these CVEs can be mitigated using memory safety techniques that prevent memory overflow. However, memory safety techniques alone cannot mitigate the rest of the CVEs, as 21% of them involve arbitrary code execution, and 61% involve both arbitrary code execution and memory overflow vulnerabilities. Therefore, enhanced security primitives are required to prevent a significant number of severe vulnerabilities.

1.1 The Unikernel Operating System Model

Unikernels [57, 82, 84, 86, 87, 95, 98, 99, 110, 144] are lightweight OSs designed specifically for cloud systems and run atop a hypervisor in separate VMs. They are, by design, capable of running only a single application. In contrast to general-purpose full-featured OSs, in unikernels, a single application is statically compiled together with the minimum necessary kernel code and libraries to produce a single address-space image – a form of library OS [48]. In contrast, the traditional VM model includes a full-featured guest OS, which includes a large number of subsystems that may not be used by guest applications. Figure 1.3 illustrates this contrast. Examples of unikernels include HermitCore [87], MirageOS [95, 96], rumprun [82], HermiTux [105], Lupine Linux [86], and IncludeOS [57], among others.

The unikernel model has several advantages. Since only select OS components are included with the application, their code size and memory footprints are significantly smaller, which also reduces their boot times. (This is potentially important if fast failure recovery through VM restarts is desired.) Figure 1.4 compares the image size, boot times, and system calls of the Kite [100], an unikernelized driver domain proposed in this dissertation, against Ubuntu Linux. In this figure, Kite encapsulates a rumprun-based [82] network driver domain that

[1]The Common Vulnerability Scoring System (CVSS) is a method for describing the severity of a vulnerability using a single numerical score [40]. Higher the number, the greater the severity.

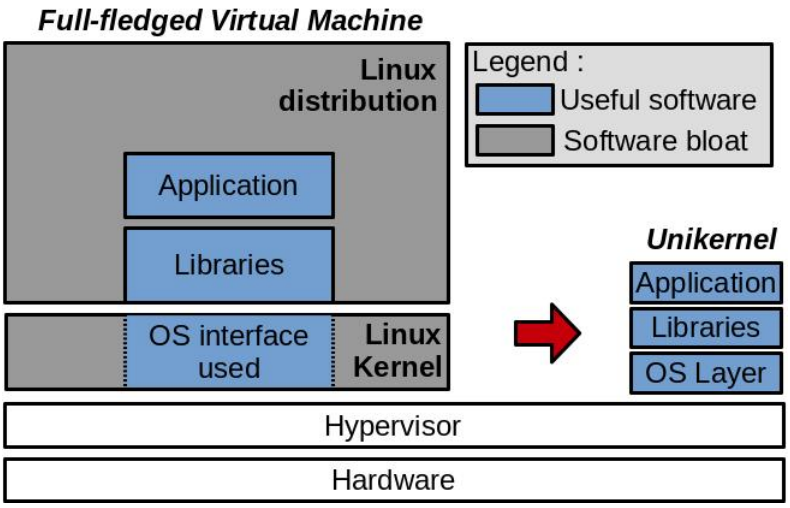
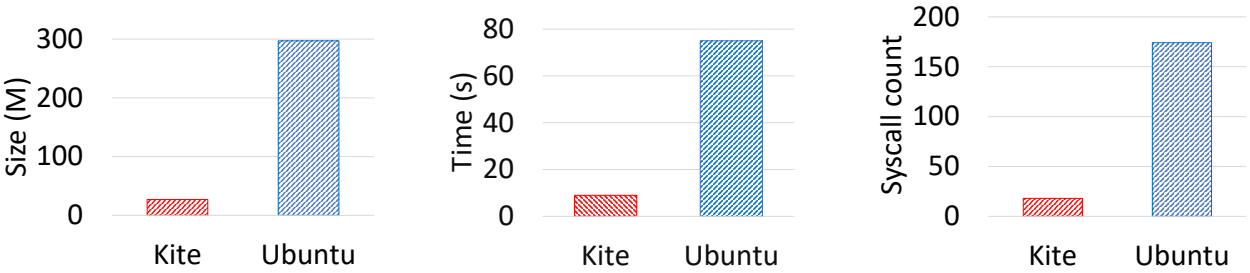


Figure 1.3: Traditional VM model versus the unikernel VM model.

we designed (Chapter 6) and implemented (Chapter 7), and Ubuntu Linux does not run any application. This figure reveals rumprun's significant savings on memory image size and boot times – by one order of magnitude.



(a) Image size. (b) Boot time. (c) System calls.

Figure 1.4: Kite and Ubuntu network domain: image size, boot time, and syscall comparison.

Unikernels also have a performance advantage. Since OS components are compiled together with the application in a single address space, system calls are converted to common function calls, a feature that can reduce system call latency by up to one order of magnitude [105], leading to performance improvement of up to 33% for OS-intensive applications [86].

Unfortunately, unikernels are designed only for user-space applications and are unsuitable for driver domains. In this dissertation, we explore the use of unikernels for driver domains.

We use the rumprun unikernel [82] for this purpose. The key feature of rumprun is that it is directly based on NetBSD OS's code [28], which makes it possible to leverage NetBSD's large collection of device drivers, including highly specialized ones such as Amazon ENA [9]. In fact, NetBSD device drivers can run inside a rumprun unikernel instance out-of-the-box.

Compared to Linux, a rumprun unikernel image is minimalistic: it can be as small as 22MB. This small code size yields an important security advantage: potentially fewer security vulnerabilities. Figure 1.5 compares the number of ROP gadgets [119], measured using the Ropper tool [24], for rumprun, vanilla Linux kernel with the default configuration, CentOS 8, Fedora Rawhide (05/2020), Ubuntu 18.04, and Debian 10.4 kernel images with their associated kernel modules. The number of ROP gadgets is one quantitative metric that is often used to evaluate security vulnerability [70]: smaller the number of gadgets, the greater is the difficulty for an attacker, broadly described, to find appropriate gadgets to launch an exploit[2]. Figure 1.5 demonstrates that rumprun has a substantially smaller number of gadgets than any of the Linux configurations. Assuming that NetBSD's code quality is on par with that of Linux, this figure also indicates the potential for improved security more generally, as rumprun's attack surface is also proportionally reduced compared to that of full-featured OSs.

Recent versions of rumprun [103] were extended to support multi-core systems and hardware-assisted virtualization in Xen, which makes it even more attractive as a driver domain OS. However, using rumprun as a driver domain OS requires overcoming several significant challenges. Xen's PV I/O support is very minimalistic in rumprun. Rumprun lacks Xen backend drivers. Thus, rumprun can be used only as a regular unikernel, but not as a driver domain. Moreover, NetBSD cannot run driver domains in Xen.

[2]We only use the number of ROP gadgets as a simple indicator of the likelihood of security vulnerabilities. Recent works [58] have demonstrated that other metrics (related to ROP gadgets) are more robust indicators. We defer such metrics to Chapter 10 and 14.

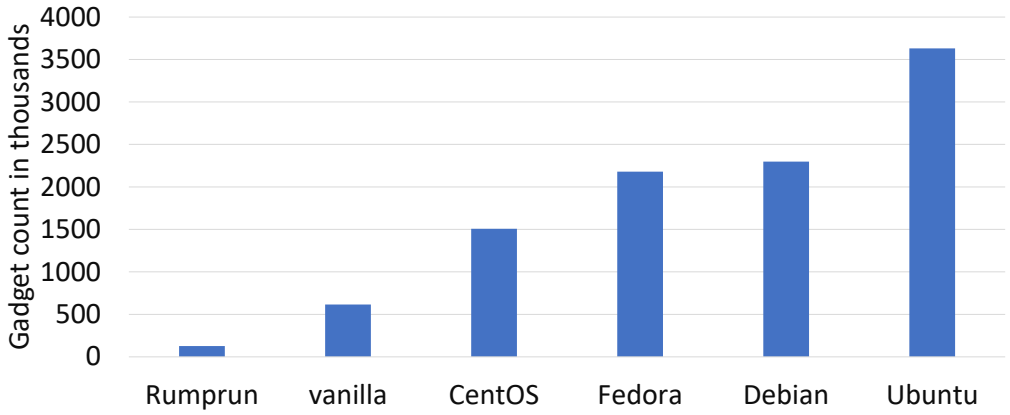


Figure 1.5: ROP gadget comparison. For a detailed gadget breakdown of rumprun/Kite and advanced gadget metrics, see Chapters 10 and 14.

We overcome these challenges developing a rumprun-based, driver-domain OS called *Kite* [100]. We instantiate Kite for the two most used devices, storage and network, by designing and implementing the storage backend (`blkback`) and network backend (`netback`) drivers, respectively. To understand the effectiveness of our implementations, we conduct extensive experimental studies using well-known benchmarks and applications. Our results reveal that our rumprun driver domains provide competitive performance to that of a Linux-based driver domain, while retaining all the benefits of unikernels such as reduced number of ROP gadgets, smaller image and code sizes, and faster boot time.

1.2 VMM's Security Threats in Clouds

Modern cloud applications have a range of safety and performance requirements. However, a rigid VMM design and configuration for every VM may not provide effective safety in many scenarios. For example, the QEMU emulator provides a floppy drive controller (FDC), even though most workloads do not use floppy devices nowadays. The VENOM vulnerability [7], which is present in the FDC may allow a guest user to crash the VM or execute arbitrary

code on the host with the privileges of the QEMU process, regardless of a VM's need for this device. This means that a faulty implementation can pose a significant security risk to the entire virtualization ecosystem.

Therefore, it is crucial to have a flexible per-VM specialization for the VMM in order to minimize the attack surface. Although various VMMs exist with a variety of features, they lack the ability to be easily customized. This causes VMs to use generic VMMs with many irrelevant default features, resulting in a larger attack surface. To overcome this problem, it is essential to design and develop a modular and configurable VMM that minimizes the TCB in cloud infrastructures. Creating such a VMM from scratch is challenging because it requires substantial expertise and engineering effort. On the other hand, the rigid design of generic VMMs makes it difficult to achieve a customized, minimal VMM for specific VMs.

Even if unwanted features are removed by specializing a VMM, it does not ensure that the VMM is safe from attacks because of existing bugs in its remaining implementation. The lack of security primitives in current VMMs to enforce separation between components makes them susceptible to exploitation. Therefore, an attacker can manipulate one part of the VMM by exploiting a vulnerability in another part. Moreover, since VMMs run as userspace processes on the host, attackers can exploit bugs in a VMM to gain the VMM process's privilege to harm the host OS and other VMs.

By using isolation mechanisms to compartmentalize unrelated modules, it is possible to prevent unauthorized access and restrict the damage caused by a security breach. Implementing this type of compartmentalization in a VMM requires careful design and implementation. Determining when to grant and restrict access to compartments requires a thorough evaluation of the interactions between different parts of the VMM.

CPU manufacturers and researchers are introducing new isolation mechanisms to improve

or replace traditional ones [35, 67, 69, 90, 124, 129]. Some of these techniques prioritize security at the expense of performance [77, 85, 117, 131], while others focus on boosting performance over security. Furthermore, not all isolation techniques are appropriate for use within a VMM. The challenge lies in finding an isolation technique that can achieve the necessary level of compartmentalization within a VMM without sacrificing the performance of VMs.

To overcome these security concerns and mitigation challenges, we have developed methods for compartmentalizing and specializing VMMs in order to improve cloud security. Our prototype, called *Redwood*, extends Cloud Hypervisor [45], an open-source VMM written in Rust, for running modern cloud workloads. We have identified thirteen crucial components of Cloud Hypervisor, such as virtual devices, and compartmentalized them in Redwood using Intel memory protection keys (MPKs) [67] on x86 servers. This isolation helps prevent unauthorized access to virtual devices or limits attackers' ability to exploit vulnerabilities to compromise other devices or execute arbitrary code. Redwood has fifteen fine-grained modules, each representing a single feature, making it highly configurable and flexible. As a result, a VMM image size can be significantly reduced at build time by including features that are necessary only for running a particular VM. Furthermore, we extend Redwood's support for running unikernels.

1.3 I/O Acceleration for VMs

I/O acceleration architectures, such as Data Plane Development Kit (DPDK) [128] and Storage Performance Development Kit (SPDK) [123], can enhance the performance of VMs with high throughput when used with VMMs. These architectures involve moving the data plane from the VMM to a separate userspace DPDK or SPDK application on the host. This

technique allows for increased I/O performance but poses a security risk, as the VMM must share sensitive information about its VMs with the accelerated DPDK or SPDK application.

Thus, the accelerated application can become an appealing hotspot for attackers, as it may contain information of multiple VMs. To mitigate this risk, we propose implementing a solution that isolates each VM's information within the accelerated application. This way, even if an attacker were to compromise the accelerated application, the information for each VM would remain protected.

1.4 Summary of Research Contributions

The dissertation's first contribution is a unikernelized driver domain OS called Kite, which is instantiated for the two most basic and necessary devices: storage and network. The dissertation's second contribution includes new VMM design principles for improving cloud security, and specialization and compartmentalization techniques that realize these principles. These principles are demonstrated by constructing a prototype called Redwood. The dissertation's third and final contribution includes isolation techniques for mitigating exploits that leverage vulnerabilities in I/O acceleration architectures used for VMs.

We now summarize these contributions.

1.4.1 Kite: Unikernelized Storage Domain

Most guest VMs need storage device drivers. PV storage drivers are one of the most common drivers used by guest VMs because of faster I/O processing than emulation. We present the Kite storage domain, an effort to design an unikernelized storage driver domain, aiming for security advantages while preserving performance benefits.

Kite uses the rumprun unikernel, which allows leveraging physical storage drivers from the NetBSD OS. However, a PV storage device needs a `blkback` driver in the driver domain to communicate and transfer data between the physical storage and the guest VM through its storage frontend (`blkfront`). We design and implement the `blkback` driver. Our experimental evaluations reveal that the Kite storage domain is as performant as Linux, one of the most widely used commodity OS for servers and desktops. Importantly, the Kite storage domain's exposed attack surface is only a fraction of a counterpart Linux domain.

1.4.2 Kite: Unikernelized Network Domain

Networking is one of the widely used functionalities for guest VMs. PV networking is the most efficient method for multiplexing one network device between several guests. We design and implement a rumprun-based Kite network driver domain, which works as a backend for PV networking and routes networking traffic from guest machines to the physical network interface card (NIC). We leverage the NIC drivers from NetBSD but implement a `netback` driver, which rumprun lacks.

The `netback` and `netfront` (network frontend) communicate using two shared memory ring buffers for receiving and transmitting network packets. Since rumprun lacks rich work queues and interrupts, we develop a multi-threaded model that helps efficiently handle network packets with no performance penalty. We also design and implement network bridging support for `netback`, which is necessary for forwarding packets from multiple guest OSs to the physical network adapter. Our evaluations show that the Kite network domain is as performant as Linux while exposing only a fraction of a corresponding Linux domain's attack surface.

1.4.3 Redwood: Flexible Secure VMM

VMMs play a crucial role in deploying and managing VMs, providing isolation and security for VMs running on the same physical machine. However, some parts of the standard VMM architecture and implementation are prone to vulnerabilities. We present Redwood, a method for customizing and compartmentalizing VMMs for localizing exploits that may leverage such vulnerabilities. We have developed Redwood's prototype based on the open-source VMM called Cloud Hypervisor. It converts the Cloud Hypervisor into a flexible VMM and uses the Intel MPK, a hardware feature, to compartmentalize virtual I/O device instances within the VMM.

Our security analysis includes a CVE analysis and attack surface analysis of the VMM, showing that Redwood can mitigate various existing CVEs. Additionally, Redwood's basic configuration exhibits a 50% smaller VMM image size, a significant reduction in quality and quantity of gadgets (measured with ropper [24, 25] and GSA-based [37, 58] metrics) compared to Cloud Hypervisor. The other three Redwood configurations are also significantly compact. Our evaluations reveal that Redwood is as performant as the original Cloud Hypervisor in most scenarios.

1.4.4 Vhost User Compartmentalization in DPDK

The DPDK library is employed in conjunction with the OVS application, known as OVS-DPDK, to enhance the networking capabilities of VMs through the vhost user interface. This architecture significantly improves the performance of paravirtual network connections. However, as OVS-DPDK can access the memory of multiple VMs, it can be exploited, causing leakage of VM data. We develop a compartmentalization technique that isolates sensitive VM information within the OVS-DPDK application. We develop a prototype implementation of

the technique utilizing the Intel MPK hardware primitive to protect VMs' memory.

Our security analysis including CVE analysis demonstrates that the compartmentalization technique can mitigate potential attacks on the OVS-DPDK application. Our performance evaluation reveals that the technique does not cause any noticeable performance degradation.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows.

Chapter 2 discusses the relevant background information that is necessary to understand the dissertation's research contributions. These include hypervisors, their PV I/O driver models for network and storage devices, VMMs, the vhost user interface in OVS-DPDK, rump kernels, the rumprun unikernel, and memory-based isolation techniques.

Chapter 3 summarizes related work on hypervisor and VMM disaggregation techniques, unikernels, driver domains, and protection key-based memory isolation techniques, and compares and contrasts them with the dissertation's contributions.

Chapter 4 discusses the challenges associated with unikernelizing the storage driver domain and the design of the Kite storage domain. Chapter 5 describes the Kite storage domain's implementation.

Chapter 6 discusses the challenges of unikernelizing the network driver domain and presents the design of the Kite network domain. Chapter 7 discusses its implementation.

Chapters 8 and 9 evaluate the Kite storage domain's and network domain's performance, respectively. Chapter 10 evaluates the storage and network domains' security benefits.

Chapter 11 describes Redwood's design principles for improving cloud security in VMMs.

This chapter also discusses the necessary changes to implement these principles through per-VM specialization and intra-VMM compartmentalization. Chapter 12 presents Redwood's prototype, including its implementation for a highly flexible and Intel MPK-based compartmentalized VMM, and compartmentalization in OVS-DPDK. Chapters 13 and 14 evaluates the performance and security of Redwood and the compartmentalized OVS-DPDK.

Finally, Chapter 15 concludes the dissertation and identifies future work in each problem space of the dissertation's contributions.

Chapter 2

Background

This chapter discusses concepts relevant to the design and implementation of Kite and Redwood. It touches upon hypervisors, such as Xen and KVM, their virtualization approaches, and the role of VMMs. This chapter also discusses unikernels, rumprun architecture, I/O accelerators, and memory-based isolation techniques.

Sections 2.1 and 2.2 discuss two popular hypervisors, Xen and KVM, architectures. Xen's I/O device model, including the network and storage drivers, are discussed in Sections 2.3, 2.4, 2.5 followed by a discussion on Xen driver domain in Section 2.6. Section 2.7 discusses the concept and different roles of VMMs. Section 2.8 discusses virtio device models, often facilitated by VMMs in conjunction with hypervisors like KVM. The relationship between vhost user devices and DPDK is sketched in Section 2.9. We then discuss the concept of unikernels, followed by a discussion of using unikernels for the driver domain and the rationale for choosing the rumprun unikernel in Sections 2.10 and 2.11. Finally, Section 2.12 discusses different techniques of memory-based isolations.

2.1 Xen

Xen is a popular open-source hypervisor [83], which pioneered a concept of "paravirtualization'' by leveraging protection rings of x86-32. These CPUs historically lacked virtualization capabilities [112], and Xen's approach was to modify the target OS kernel so that it

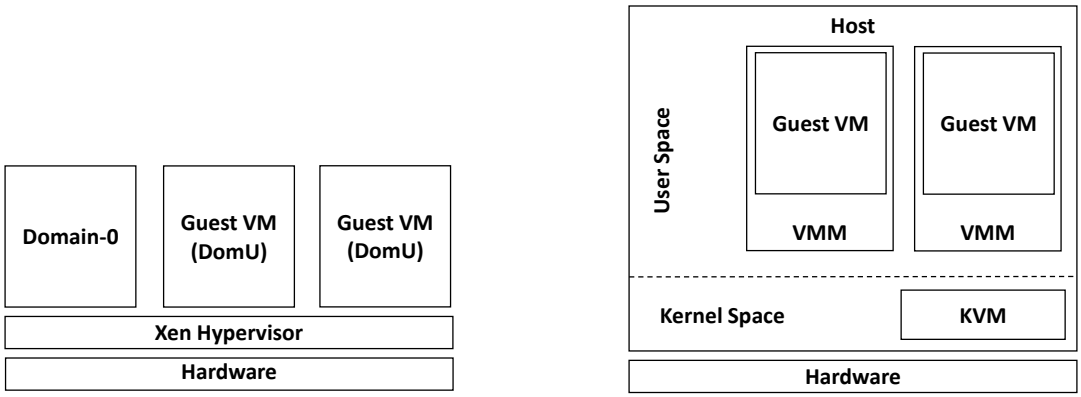
runs on top of a hypervisor. Later, CPUs enabled hardware-assisted virtualization support through VT-x and AMD-V (or other extensions by non-x86 vendors). *Hardware Virtualization Mode* (HVM) was proposed and widely adopted by Xen since then. HVM is also preferable nowadays since it is not affected by recently discovered security vulnerabilities, such as Meltdown [91]. Figure 2.1a depicts Xen's architecture. Xen has implementations for VM creation, memory management, scheduling, etc. The VM that runs administrative OS is known as Dom0, used for running Xen utilities, such as *xl* and *Xenstore*, and device drivers. The guest VMs in Xen are called DomU.

2.2 KVM

KVM (Kernel-based Virtual Machine) [76] is another open-source hypervisor that uses Linux-based OSs. KVM is built as a kernel module for Linux, which makes it highly efficient and secure. The Linux kernel that runs on bare-metal and loads the KVM module is known as host OS. The virtualization is achieved through hardware virtualization support (such as Intel VT or AMD-V) on the host system. To provide a complete virtualization solution KVM relies on the host Linux's device drivers, memory management, and process management for resource distributions. KVM also needs help from VMM, which runs as userspace process, for managing guest VMs.

2.3 Xen I/O Drivers

Traditional I/O device emulation is inefficient due to substantial performance overheads [59]. Xen's original paravirtualization method proposed to handle I/O through special PV drivers. PV drivers are also used in the HVM mode as long as the corresponding guest OS is *enlight-*



(a) Xen architecture.

(b) KVM architecture.

Figure 2.1: Xen hypervisor runs on bare metal hardware, and guest VMs run directly on the hypervisor layer, where the administrative OS also runs inside a guest VM. Xen has its implementation for resource distribution and VM management. KVM module converts a Linux (Host) into a hypervisor, where guest VMs are managed with the help of VMM inside the host OS that distributes resources between VMs.

ened about Xen's presence. Other hypervisors, e.g., VMware, Hyper-V, and KVM, similarly implement faster I/O drivers.

PV drivers in Xen are typically divided into two parts: *frontend* and *backend*. The frontend driver runs in the guest OS, denoted as DomU. The backend driver runs either in Dom0, the very first (privileged) guest, or in a dedicated driver domain (see Section 2.6).

2.4 Xen Blkfront and Blkback

The PV storage frontend (*blkfront*) provides an interface to abstract secondary storage. Therefore, the other part of the OS and applications can use this interface to issue regular block device operations such as read, write, etc. On the other hand, the PV storage backend (*blkback*) is responsible for performing these operations on the real device using the physical storage device driver. A storage domain should be able to support multiple blkfront from the same or different guest machines as shown in 2.2.

Both ends of the PV driver get to know each other's configurations using the Xenstore database. The communication between the blkfront and blkback is maintained using Xen's shared memory and ring buffer mechanism. The blkfront allocates the shared memory and a ring buffer, which are used by both ends to transfer data. Using the ring buffer, the blkfront sends requests with information, like sector number, size, etc., to the blkback for performing specific kinds of operation on the storage device. On the other hand, the blkback sends responses upon performing such operations. The storage data is transferred between these ends using direct memory access (DMA) transfers, known as the grant table mechanism.

2.5 Xen Netfront and Netback

In the case of network interface card (NIC), the frontend (*netfront*) exposes a virtual network interface to the network stack of DomU, whereas the backend (*netback*) talks to the actual physical NIC. The netfront and netback drivers establish a connection between each other using Xen-specific mechanisms such as *Xenstore* and *Xenbus*. Netfront and netback transfer data between each other using shared memory ring buffers. As a result, DomU can send packets to the physical NIC through netfront.

For each netfront, there should be one corresponding netback as shown in Figure 2.2. The netback and netfront drivers use two shared ring buffers for data exchange, which are allocated by netfront. One ring buffer, Tx, is used for sending packets from netfront to netback. Another ring buffer, Rx, is used for sending packets from netback to netfront.

As shown in Figure 6.1, there are multiple netbacks with multiple DomUs. One way to share the same physical NIC is to use *bridging* by connecting all netbacks to a bridge. The network bridge, which is connected to the physical NIC, routes packets between netbacks and the physical NIC as well as across different netbacks. Each netback is assigned its own

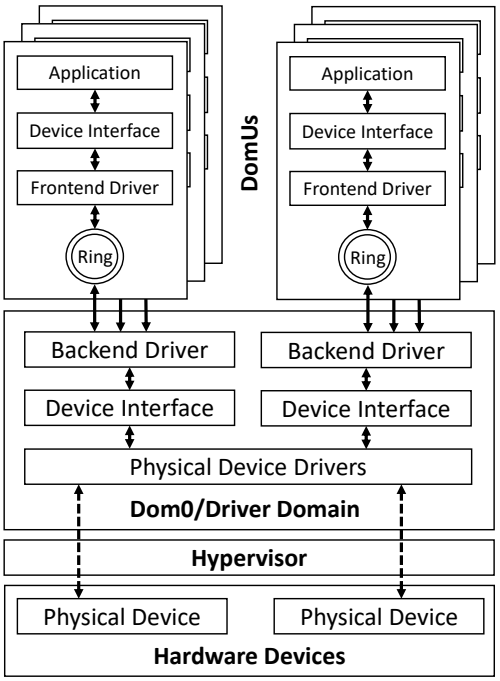


Figure 2.2: Xen's PV I/O device driver model.

IP address on the network.

2.6 Xen Driver Domain

Xen's privileged Dom0 domain runs device drivers and performs many critical system tasks. To offload Dom0, *driver domains*, special unprivileged guest OSs which run device drivers, are often used. Driver domains also increase isolation and overall system security since potentially vulnerable drivers are isolated from Dom0. Driver domains have direct access to the underlying hardware by using PCI passthrough capabilities of the hypervisor. They are more typical for networking, where the netback driver runs inside a separate VM rather than Dom0. Although storage also has corresponding blkfront-blkback I/O drivers, they seem to be deployed in Dom0 for typical setups.

The use of driver domains is exemplified by Qubes OS [114], an OS which runs individual Linux instances for each group of applications in VMs atop of Xen to provide very strong security. Qubes OS uses network driver domains to strongly isolate network device drivers from the system. For stronger isolation, Qubes OS also takes advantage of I/O memory management unit (IOMMU) [47, 79], which we also support in our design. Full-fledged IOMMU support needs HVM; it safely remaps interrupts and memory addresses to protect against both malicious (or faulty) devices and vulnerable (or buggy) device drivers.

By moving device drivers to a separate domain, Dom0's attack surface reduces. As a result, any error or exploit in the corresponding code will not affect Dom0, and the Xen administrative interface to other guest OSs remains uninterrupted. Though other open-source virtualization technologies such as KVM support faster I/O, they do not yet implement driver domains, unlike Xen.

In this dissertation, we propose to use a unikernel for Xen driver domains. By using a light-weight unikernel, we are able to reduce memory footprints as well as reduce the attack surface of driver domains.

2.7 Virtual Machine Monitor

A VMM is a software run in host OS which plays vital roles in virtualization technology. There are many well-know VMMs such as QEMU, Firecracker, Cloud Hypervisor, kvmtool, etc., that work with KVM. The main roles of a VMM include:

- **Resource abstraction:** A VMM helps provide virtual resources to guest VMs. It abstracts the physical resources of a host machine, such as CPU, memory, and I/O devices, with the help of a hypervisor and hardware virtualization support.

- **Resource allocation:** A VMM dynamically allocates physical resources to guest OSs to serve the needs of each VM. With the help of a hypervisor and host, a VMM manages the use of resources across multiple VMs and ensures that each VM gets the resources it needs to run efficiently.

- **Isolation:** It is an essential feature to run multiple VMs on the same physical machine without risking one VM affecting the others. VMMs share responsibility for providing isolation between VMs so that each VM runs in an isolated environment.

- **Security:** Similarly, VMMs should enforce access controls and security policies so that it is possible to provide security to run sensitive applications in VMs, without risking security breaches or data theft.

- **Performance:** VMMs use hardware and software acceleration and other performance optimization techniques to enable high performance virtualization. They reduces the virtualization overhead to run VMs with near-native performance.

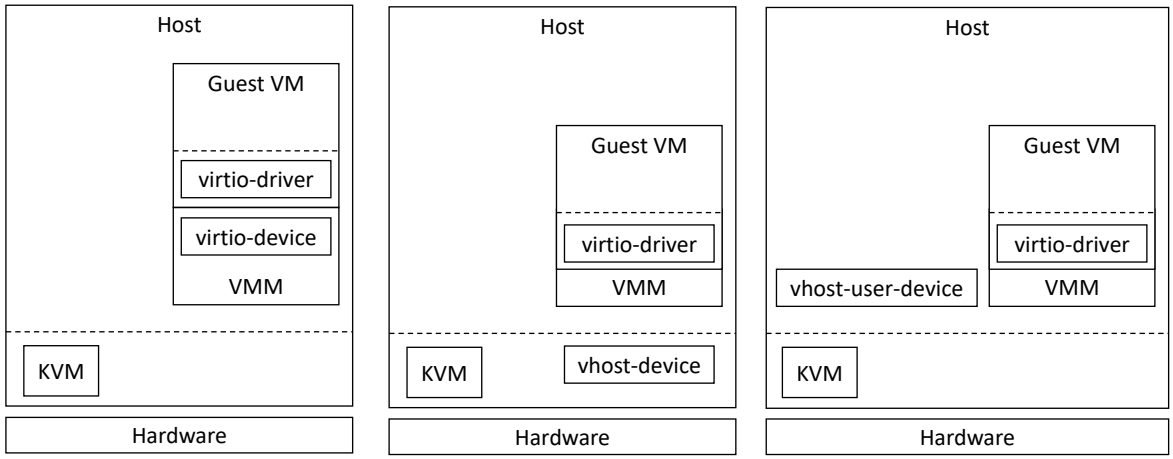
- **I/O virtualization:** VMMs offer emulated and PV I/O devices to the guest VMs, making it possible to share I/O resources between multiple VMs and improve VMs' I/O performance. There are multiple I/O schemes. Some of them are discussed in 2.8.

- **VM Migration:** This is necessary for high availability, disaster recovery, and load balancing. Some VMMs support migration, which allows VMs to be moved from one physical host to another without downtime.

- **Snapshots:** It allows users to save the state of a VM at runtime and restore it later if needed to resume the VM from that state. Snapshotting can be helpful for testing, development, and bug detection when testing new software or configurations.

2.8 Virtio PV Devices

A major role of a VMM is managing I/O for VMs. Emulating traditional I/O devices can be inefficient due to significant performance penalty [59]. To improve performance, VMMs offer special PV I/O devices. Virtio [113, 130] is a specification for PV devices that provides an interface for software to manage and exchange information. PV devices can be exposed to the emulated environment through PCI, memory mapping I/O, and S/390 channel I/O. Some communication tasks, such as device discovery, are delegated to these methods.



(a) Virtio device. (b) Vhost device. (c) Vhost-user device.

Figure 2.3: Different modes of virtio devices.

A *virtio device* runs on the host and translates signals between the *virtio driver* and the virtio data plane. It can handle both physical devices (like a NIC) and virtual devices (like a virtual NIC). The virtio driver in the VM communicates with the virtio device according to the virtio specification. It performs tasks such as detecting a device, allocating shared memory for communication, starting the device using the virtio protocol, and maintaining communication with the virtio device. The virtio-device and virtio-driver concept is inspired by Xen's PV backend and frontend, respectively.

The device and driver communicate using virtqueues and notifications. Virtqueues are queues of guest buffers that the host reads from or writes to, then returns to the guest. The memory of these queues is arranged in a circular ring known as a virtring or vring. Driver notifications from the guest VM are transported to KVM interruptions via methods such as PCI, halting the guest's processor and transferring control to the host. The device notifications, on the other hand, are a type of IOCTL that the host sends to the KVM device. Information about virtqueues can be obtained by QEMU through shared memory.

There are several ways virtio devices can operate, depending on where the device's data plane is located. When the VMM handles both the data plane and control plane operations, it is referred to as a virtio device, such as virtio-net for networking. If the VMM delegates the data plane to the kernel, the device is called a vhost device, such as vhost-net for networking. When the VMM offloads the data plane to a separate user-space process on the host, it is called a vhost-user device, such as vhost-user-net for networking. Figure 2.3 depicts these variations. Each has its benefits and drawbacks in terms of security and performance, but these discussions are beyond the scope of this dissertation. A modern VMM should support all variations so that workloads can choose the one that best meets their needs.

2.9 OVS-DPDK Vhost User

Open virtual switch (OVS) is an open-source software switch designed for virtualized environments. To eliminate the context switching between kernel and userspace while exchanging packets between two userspace entities connected through the switch, OVS can run its forwarding table in userspace.

DPDK is a library that helps speed up packet processing in data plane applications. It works by allocating resources to dedicated logical processing cores before the data plane

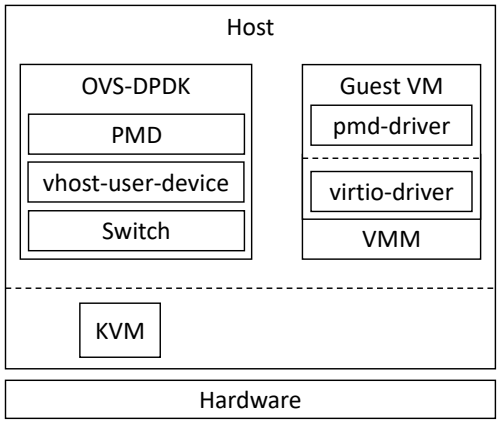


Figure 2.4: OVS-DPDK architecture.

application is called. In contrast to a Linux kernel, which uses a scheduler and interrupts to switch between processes, DPDK accesses devices through constant polling. This avoids the overhead of context switching and interrupt processing. DPDK's poll mode drivers (PMD) allow packets to be transferred directly between user space and physical interfaces, bypassing the kernel network stack entirely. This bypassing of the kernel stack and elimination of interrupt handling can provide a significant performance boost.

It is possible to link an OVS application with DPDK, which offers vhost-user APIs. Therefore, a VM can connect to an OVS-DPDK application, running in userspace on the same host, through the vhost-user interface. For each guest VM created on the host, the OVS-DPDK application can instantiate another vhost-user backend to communicate with the guest's virtio driver. As a result, OVS can forward packets between VMs on the same machine without the overhead of context switching. Furthermore, OVS-DPDK can be configured to use PMD on the host, and similarly, applications in guests can use DPDK PMD with their virtio driver to boost performance. Figure 2.4 shows the OVS-DPDK architecture, where OVS-DPDK runs as a userspace process, serving vhost-user-devices to multiple guest VMs.

2.10 Unikernel

Unikernels [57, 82, 84, 86, 87, 95, 98, 99] are a type of lightweight OS that is optimized for use in cloud systems and runs on top of a hypervisor in a separate VM. They are designed to run a single application and are created by statically compiling the application with the minimum necessary kernel code and libraries into a single-address-space image. This reduces the code and memory footprint, making the attack surface smaller, and eliminates the need for context switching, allowing system calls to be made as ordinary function calls, which can improve performance. Despite their minimalistic nature, unikernels are run by full-fledged VMMs like QEMU. To embrace minimalism, our prototype minimal VMM includes support for unikernels.

2.11 Rump Kernels and Rumprun

For the network and storage driver domains, we need a unikernel capable of running many device drivers. A fair number of non-compatible network (such as Ethernet [88], Wireless LAN [44], etc) and storage (such as PATA, SATA, SCSI, and NVMe) controllers must be supported. Because porting incurs non-trivial engineering effort, we need a unikernel that can reuse existing drivers.

NetBSD [28], a well-known general-purpose OS, has a unique property in that all its core kernel components are refactored into *anykernel* components. The anykernel concept implies that these components can be used in any context, e.g., a device driver can be executed in a user thread. A special *rump kernel* glue layer enables the reuse of the anykernel components outside of the NetBSD kernel.

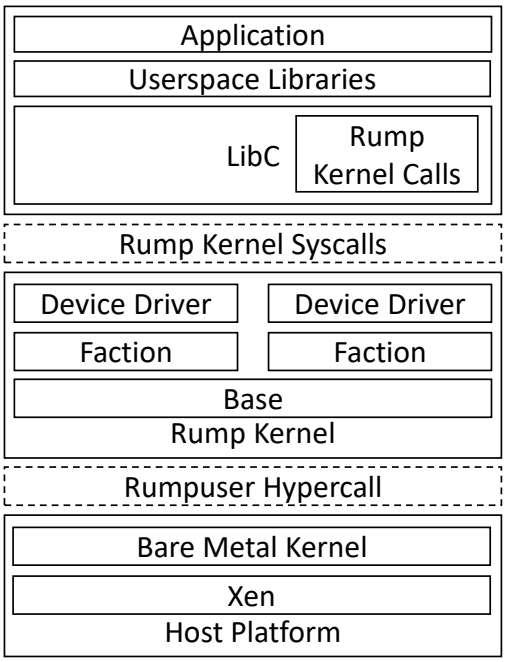


Figure 2.5: Rumprun stack on Xen.

Rumprun is a unikernel that leverages rump kernels such that it can potentially reuse any NetBSD device driver. Figure 2.5 shows the rumprun software stack, which consists of the platform-specific layer (Xen) and *bare metal kernel* (BMK) layer, which implements thread management, scheduling, interrupts, and memory management. A special *rumpuser* layer implements an interface (known as "hypercalls", which are not to be confused with Xen's hypercalls) for the rump kernel components to communicate with the BMK layer. This dissertation reuses other NetBSD components, such as the TCP stack and vnode block device interface that are denoted as 'Faction.'

The layers above the rump kernel consist of relevant libraries and their interface to the rump kernel. A unikernel application runs on top of the stack. NetBSD system calls from LibC are replaced with ordinary function calls. Since drivers need semantically similar support routines they use in NetBSD, the rump kernel contains 'Base' which provides support for memory allocation, thread handling, and locking.

Although a number of embedded Linux systems exist, we are not aware of a comparable minimal system that can readily be used for driver domains. Linux-based unikernels [86, 111] currently lack maturity and flexibility. In contrast, rumprun is stable, and rump kernels are upstreamed to NetBSD.

2.12 Memory-based Isolation

Memory-based isolation techniques aim to improve software security by isolating different parts of a program's memory, preventing them from interfering with each other and reducing the attack surface. Therefore, these techniques can help prevent data breaches and minimize the impact of security vulnerabilities [64]. Some of them are discussed below.

- Memory segmentation: Isolated memory in segments. Each segment is given specific permissions and access controls to prevent unauthorized access and data leakage.
- Virtual memory: Widely used technique in OSs - provides an abstraction on the physical memory by creating multiple virtual memory spaces for different parts of a program. Each part is allowed to access only its own allocated memory space. Hardware and software supports are leveraged for physical to virtual memory mapping.
- Containerization and sandboxing: These techniques use virtual environments to isolate applications and their dependencies. A separate environment for each application reduces the risk of data leakage or interference.
- Process isolation: Each program runs as a separate process, with its own memory space, to prevent one program from interfering with another. Often one program is divided into multiple processes to enhance isolation, where processes use inter-process communication or remote procedure calls to synchronize between different parts.

Each technique has associated advantages and disadvantages. We explore memory-based techniques for inter-process isolation as this dissertation compartmentalizes VMM components. The software fault isolation (SFI) [133] technique uses memory-bound checks to prevent one component from indirectly accessing the memory of another. It requires code instrumentation at compile time or binary rewriting. The execution overhead of SFI techniques is up to 42% [85, 117].

Some approaches use hardware page protection for memory isolation [50, 54, 63, 89, 92, 93]. As part of the address translation, hardware checks the access permission, which incurs no additional overhead on execution. However, changing control between two components involves a base change in the extended page table (EPT) that requires expensive context switching. Though light-weight context switching approaches [54, 63, 78, 92, 93] reduces overhead, the toll is still significant, where the upper limit varies from 10% to 65% [92, 131], depending on the application and approach.

Chapter 3

Related Work

Several state-of-the-art works share motivations with our work in some aspects. Moreover, different existing approaches previously attempted to secure virtualization platforms in different ways. On the other hand, unikernels are being used on different hypervisors and proposed to solve various interesting problems. Similarly, various compartmentalization techniques are being exercised to improve security in various platforms. In this chapter, we discuss some existing relevant works on hypervisor and VMM security and unikernel usages.

The discussion of various existing disaggregation techniques for enhancing the security of virtualization environments is presented in Section 3.1. In Section 3.2, the benefits of using unikernels in improving cloud infrastructure are discussed. Prior approaches to enhance driver domains are examined in Section 3.3. Finally, Section 3.4 explores works that leverage protection keys to benefit from memory-based isolation. Tables 3.1and 3.2 summarize works related to approaches proposed by Kite [100] and Redwood, respectively.

3.1 Hypervisor Disaggregation Approaches

Hypervisors, which make it possible to run multiple VMs on the same physical machine, are counted towards a TCB for cloud infrastructures. The Xen hypervisor uses Dom0 as a control VM. Dom0 is a fully-fledged OS that runs on top of Xen. Unexpected behavior from Dom0 or Xen can immediately and adversely affect any (DomU) guest OS. Therefore, there

Table 3.1: Prior arts related to Kite's approach.

Approach	Isolation type	Disaggregate	Driver Domain	OS type
Xoar [65]	VM	✓	✓	Full-fledged
QubesOS [114]	VM	✓	✓	Full-fledged
Nexen [120]	Memory	✓	✘	Full-fledged
Murray et al. [102]	VM	✓	✘	Lightweight
LibrettOS [103]	VM	✘	✘	Lightweight
Docker [101]	Process	✘	✘	Lightweight
Kite	VM	✓	✓	Lightweight

have been several efforts [65, 102, 120] for splitting Xen responsibilities, so that an exploited or failed component does not affect other components.

Xoar [65] disaggregates Dom0 functionality into nine types of service VMs, each having different responsibilities. Two of them, PCI backend and bootstrapper, run on top of nanOS, a lightweight OS, destroyed after initialization. Among other service VMs are network driver domain and block driver domain. Driver domains execute corresponding backend drivers. Xoar allocates one backend driver for each frontend driver so that one frontend does not adversely affect another frontend. Every driver domain runs a fully-fledged OS such as Linux, which still has a potentially large attack surface.

Qubes OS [114] implements a protected Xen-based user environment using four types of VMs: apps VMs, network VM, storage VM, and administrative/graphical user interface (GUI) VM. The apps VMs are domains for running corresponding types of applications. The network VM runs netback and serves as a network driver domain for the apps VMs. The storage VM provides access to the disk for the apps VMs. The administrative/GUI VM provides GUI to users. For all types of VMs, Qubes OS runs Linux. Since Qubes OS uses standard Xen driver domains, our Kite driver domains can also be integrated into Qubes OS easily for reducing attack surface and memory footprint of driver domains. Reducing

memory overheads is crucial for desktop environments that Qubes OS primarily targets.

The NeXen [120] architecture decomposes hypervisor into three parts using paged-based isolation mechanisms: security monitor, shared service domain, and Xen slices. The security monitor provides isolation between internal domains and manages privileges by controlling all updates to the memory management unit (MMU). Xen slices are composed of highly vulnerable hypervisor functionalities and data needed by the DomU. Each slice serves only one DomU. The shared service domain provides the functionalities that could not be decomposed into slices. One limitation of this work is that NeXen does not manage I/O devices. Therefore, NeXen relies on the native Linux PV (e.g., network and disk) device drivers and cannot prevent abuses on the drivers.

Murray et al. [102] disaggregate the hypervisor by extracting the domain building process, called domain builder, from Dom0 and porting it to a light-weight OS such that the TCB attack surface remains small. However, for I/O calls, the domain builder relies on Dom0 which runs a backend driver as well as a physical driver. Therefore, this disaggregation does not secure the I/O path. SSC [61] describes a modified Xen architecture for reducing TCB by distributing DomU responsibilities to multiple user-level service domains called UDom0. Each DomU belongs to one UDom0, which enforces isolation. Apart from UDom0, this design has a system-wide administrative domain, called SDom0, and a domain builder. SDom0 has multiple responsibilities, including scheduling and I/O device virtualization. Therefore, SDom0 has a relatively larger attack surface and errors can affect core functionalities.

These approaches incur a performance penalty and were specifically designed for the Xen bare-metal hypervisor, where all the VMs, including Dom0, run on top of Xen. Because of architectural differences, these approaches do not apply to other hypervisors. For example, KVM cannot run on bare metal and instead it converts a Linux OS into the hypervisor, known as the host OS. As a result, Linux's functionality, such as the memory management

and scheduler, becomes the functionality of the hypervisor. Unlike Xen, KVM cannot run PV backends inside a separate VM. KVM runs on hardware that provides virtualization support, such as Intel VT-x, so that VMs can benefit from hardware-assisted isolation.

KVM, Hyper-V, and several other hypervisors rely on VMMs, such as QEMU [66] and Firecracker [118], for VM creation and management, I/O device emulation, and other responsibilities. It is possible to run each VM on a separate VMM process, providing process-level isolation between VMM instances. This dissertation addresses the risks associated with VMMs for VMs. While the current design does not allow for the disaggregation of the VMM, we propose intra-VMM isolation to improve security without sacrificing performance.

3.2 Unikernels for Cloud Infrastructures

Every workload in the cloud runs inside VMs. Unikernels are light-weight OSs that claimed to have various desirable properties by the researchers, which make them a good fit for the cloud infrastructure over the existing setup. In terms of boot time, some works [97, 137, 139] show that deployment of scaled cloud workload in remote data centers is faster and smoother compared to Linux. Moreover, memcached was shown to have a throughput that is twice as faster using a unikernel [116] compared to Linux.

On the other hand, because of the portability and consistency, application containerization became very popular for deployment in clouds. However, containers provide process-level isolation where unikernels themselves are VMs, often supported by hardware-assisted virtualization, which offers better isolation. To provide better isolation, containers often run inside VMs that use fully-fledged OSs [68, 72]. These OSs have a bigger attack surface than unikernels. Moreover, Goethals [74] et al. show that for microservice applications, the OSv [84] unikernel performs up to 38% faster than Docker [101]. LibrettOS [103] shows that

rumprun-based unikernels can be used as servers in a single OS. LibrettOS's NFS server is 9% faster than that of either Linux or NetBSD, and Ngnix HTTP server is up to 66% and 27% faster than NetBSD and Linux, respectively. We use and extend LibrettOS's multi-core and Xen HVM support in our work.

HEXO [106] takes advantage of the low resource requirement nature of a unikernel, named HermitCore [87]. Authors were able to offload server workloads to light-weight, low-cost embedded computer boards such as Raspberry Pi [10]. This way, they were able to improve the throughputs for compute-intensive workloads in servers up to 67%, costing negligible amount of money for infrastructure and energy.

The faster boot time, lower resource requirement, and better performance of unikernels are inspiring the researcher in the academia and industry [8, 111] to adapt unikernel architecture for improving the cloud infrastructure. However, none of these prior works attempted to improve the hypervisor itself. In this dissertation, we show how adapting the unikernel concept can benefit hypervisors and make them more secure and performant, while reducing resource consumption.

3.3 Driver Domain and Backend Improvement

Several approaches were proposed by the researchers previously to improve the Xen PV device drivers and driver domains. Sushrut Shirole came up with the idea to replace the interrupt-based event handler with a polling-based request and response handler for both front and backends [121] to improve driver domain performance. This approach requires modification to Linux frontend and backend drivers for compatibility, where our work results in a lightweight driver domain without losing any compatibility with the existing frontends.

XCollOpts [142] focuses on the existing VM scheduling scheme that supposedly favors CPU-intensive workloads over the latency-sensitive I/O workloads running in a VM. In contrast, the authors proposes premature preemption prevention techniques for driver domains and CPU load balancing-based optimizations for credit schedulers to benefit I/O virtualization. Moreover, XCollOpts suggests multiple Tx and Rx buffer pairs in the network driver domain to leverage multicore systems along with methods to reduce grant page accesses for the small-sized packets.

Bourguiba et. el [55, 56] proposes a I/O virtualization model based on packet aggregation to transfer packets between the driver domain and the VMs. Their work shows network throughput improvement at the cost of latency. To mitigate latency degradation, the proposes a dimensioning tool that helps to dynamically balance between throughput and latency in different situations.

All of the above-mentioned prior works regarding I/O virtualization and driver domain focuses particularly on performance (especially for network) rather than other important aspects such as security and resource utilization. Moreover, it seems they consider Linux as the base OS for the driver domain. One exception would be the LibrettOS's Network Server [103] that implements a network server in a unikernel. This network server runs as a network backend where the communication with the frontend is done using improvised ring buffers, which are only compatible with network clients with such ring support. Therefore, unlike our proposal, this model is not compatible with the frontends from existing commodity OSs such as Linux, NetBSD, and Windows.

3.4 Protection Key-based Memory Isolation

Intel MPK is a feature in Intel CPUs that enables memory protection using protection keys. Various memory-based isolation schemes [73, 80, 81, 90, 115, 124] leverage MPK to separate trusted and untrusted components. These schemes create separate memory protection domains by assigning different pkeys to memory pages containing different codes and data. The access of one domain to another domain's code or data depends on the protocol devised by a particular approach.

In addition to its use in memory isolation, MPK has also been employed to harden JavaScript engines [108], reinforce other exploit mitigations [60, 62, 75, 85], and provide software abstractions for isolation and sandboxing [77, 107, 131, 132]. It can also be used to implement eXecute-Only Memory (XOM), a defense against code-reuse attacks that rely on reading code [53, 109]. The XOM-Switch tool [143] patches the dynamic linker and libc to mark all pages containing executable code as execute-only using the PKU system call.

LibhermitMPK [124] introduced the concept of intra-unikernel isolation using Intel MPK, dividing the unikernel into three domains for safe kernel code, unsafe kernel code, and user code. However, MPK cannot protect the codes. Therefore, LibhermitMPK designs isolation between domains by manipulating access permission to their memory. When a domain with safe code calls a function in another domain with unsafe code, access to the current domain's memory is disabled and re-enabled upon return from that function. This way, any attempt from the unsafe domain to access the safe domain's memory will cause a segmentation fault, ensuring protection against potential harm from unsafe code.

Later, Hodor [77] advanced the concept of domain isolation to the idea of protected libraries in an application running on general-purpose OSs. Each library is kept in a separate domain of executable code, with access to certain parts of the address space granted to each domain

while access to others is restricted. Erim [131] proposed a similar concept by dividing the application's code into trusted and untrusted domains. Trusted domains can access untrusted domains' memory, but untrusted domains cannot access trusted domains' memory. The transition between domains requires a change in permissions for the corresponding memory, using keys associated with these domains. The main difference between Hodor and Erim lies in the mechanisms used to ensure that transitions cannot be hijacked, preventing attackers from arbitrarily changing permissions on the keys to execute code or memory from trusted/protected domains.

Recently, FlexOS [90] extended the idea of libhermitMPK, Hodor, and ERIM to compartmentalize unikernels into further granularity. There can be multiple compartments, each having one or more libraries where the compartments are mutually untrusting. Therefore, any transition from one compartment to another requires disabling the current compartment so that the destination compartment cannot access other compartments' memory. FlexOS's prototype has an MPK-based implementation. However, none of these approaches or techniques have previously been used for VMM compartmentalization. Moreover, one cannot directly apply these protection key-based techniques because of the difference in VMM's architecture to the unikernel or regular application, which does not compartmentalize implementations like I/O devices.

3.5 Virtual Machine Monitors

There are a several VMMs and approaches for increasing isolation between VMs. Table 3.2 shows a summary of these approaches. QEMU [66] is a heavily-used full-featured VMM, which is written in C. It is designed to be generic for supporting a wide range of VMs. It is possible to launch separate QEMU instances for each VM so that each can benefit from

Table 3.2: Prior arts related to Redwood's approach.

Approach	Isolation type	Per-VM flexibility	General OS support	Language
QEMU [66]	Process	✘	✓	C
Firecracker [118]	Process	✘	✘	Rust
Cloud Hypervisor [45]	Process	✘	✓	Rust
kvmtool [38]	Process	✘	✓	C
uKVM [138]	Process	✓	✘	C
Turtles [51]	Nested VM	✘	✓	C
No Turtles [34]	cgroups	✘	✓	C
Redwood	Intra-process	✓	✓	Rust

process-level isolation. The monolithic design of QEMU does not allow isolation inside the QEMU instance.

Firecracker [118] and Cloud Hypervisor [45] are two comparatively newer VMMs. Both are written in Rust, enabling them to enjoy security from the robust memory model enforced by this language. Firecracker is designed to run lightweight Linux VMs, called microVM, and offers a small set of virtio device implementations. Cloud Hypervisor is developed targeting cloud OSs, such as Ubuntu Cloud, to run modern cloud workloads. This VMM offers more devices than Firecracker and has some additional features like VM migration, VFIO [5], and vDPA [4]. However, both VMMs provide only process-level isolation.

The kvmtool [38] VMM is written from scratch and can boot Linux VMs. It implements several virtio devices but is not known for feature-completeness like QEMU and lacks fine-grained modularity. In contrast, uKVM [138] provides per-VM specialization but can only run unikernels. Both VMMs are written in C, are very lightweight, and offer only process-level isolation.

The turtles project [51] offers nested virtualization for KVM, enabling VM-level isolation between multiple hypervisor instances running on the same physical host. While this approach

increases isolation, multiple hypervisors increase the attack surface and exhibit poor I/O performance. Therefore, a very recent approach [34] proposes using cgroups [36], instead of nested virtualization, to run a group of VMs sharing common resources. While cgroups offer isolation between VMs, they cannot provide fine-grained intra-process isolation.

Chapter 4

Kite: Unikernelized Storage Domain

Unikernelizing a storage domain is a non-trivial problem because of several aspects related to unikernel and Xen's storage domain architecture. A careful design can provide a more compact storage domain than the state-of-the-art without losing any performance benefit. In this chapter, we discuss our design choices to achieve such qualities.

In Section 4.1, we discuss the challenges associated with unikernelizing a Xen storage domain. In Section 4.3, we discuss the adaptation of rumprun's physical storage driver from NetBSD so that the unikernelized storage domain can perform operations on the physical storage device. In Sections 4.4 and 4.5, we discuss Kite's design for the unikernelized storage domain on top of rumprun, considering the discussed challenges.

4.1 Challenges

Xen PV storage frontend and backend drivers are available in different commodity OSs, such as Linux and NetBSD. Therefore, it is easier to take one of these OSs as they are and deploy it as a storage driver domain. In contrast, no existing Xen-based unikernel has both ends of the PV storage driver implemented. To be specific, rumprun leverages Mini-OS for Xen interfaces, such as Xenbus, and has the frontend driver implementation of Xen PV storage driver. Neither rumprun nor Mini-OS has blkback implemented, and it cannot be leveraged from NetBSD because the backend is associated with platform-specific implementation. This

situation poses the challenge of the design and implementation of blkback in the unikernel context, which is fundamental for getting a unikernelized storage domain.

Now, a rumprun blkback needs some specific attention, unlike the commodity OSs, because of its *any kernel* philosophy, which allows us to use physical storage drivers from NetBSD. To access the physical device driver from NetBSD from rumprun, we need to access the storage device interfaces provided by NetBSD. At the same time, since our goal is to leverage the unikernel concept, we cannot use core kernel functionalities from NetBSD, such as memory management and scheduling. For these functionalities, we depend on the rump kernel. Therefore, it requires a careful separation and interaction between what we can leverage from NetBSD and what we cannot.

Xen provides some device-specific scripts for commodity OSs. The execution of these scripts helps the backend domain (Dom0 or driver domain) to accomplish specific operations necessary for paravirtualization. Xen invokes these scripts in Dom0 or the driver domain upon the corresponding front-end device driver's request for a backend device driver. For instance, the block script retrieves requested storage device-specific information and writes them to the Xenbus database. Rumprun is a single address space OS, which lacks the luxury of running scripts. Therefore, accomplishing these block script-specific tasks for the rumprun driver domain is another challenge.

4.2 Threat Model

Storage driver domains are potentially vulnerable to attacks from malicious actors who want to gain unauthorized access to the system's data or resources. The Kite storage driver domain shares the same threat model as that of any driver domain and any VM such as [65, 102, 120].

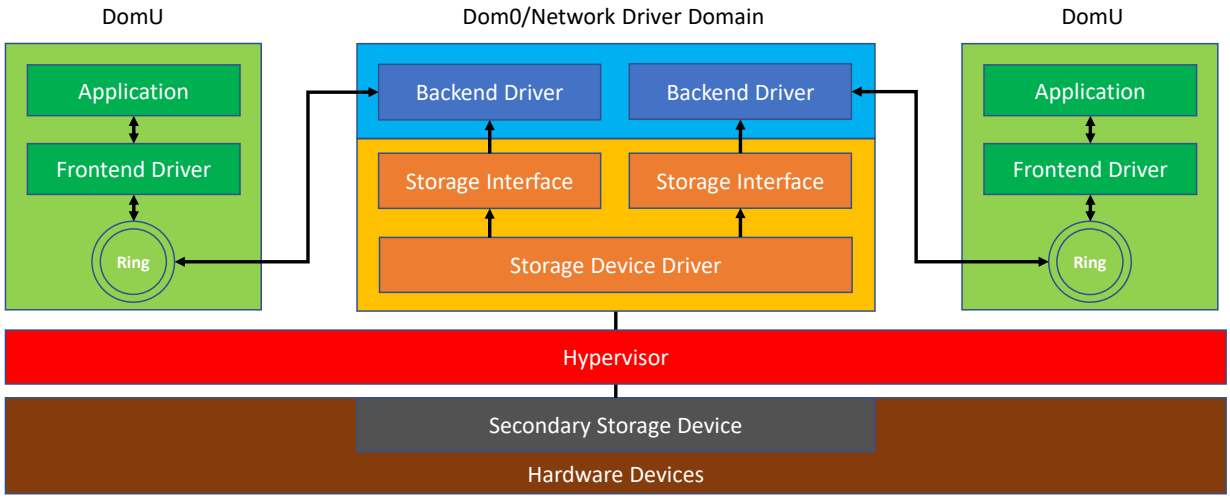


Figure 4.1: Xen's PV storage driver model.

We consider the Xen hypervisor to be the trusted component of the system that provides virtualization capabilities. The compiler is another trusted component of the system that is used to translate source code into machine code. Domain-0 is also considered trusted as it is the administrative domain on the Xen hypervisor.

Both physical and PV drivers are components of the driver domain that manage the communication between the hardware and the virtualized environment. They are potentially vulnerable to attacks from malicious actors who want to gain unauthorized access to the system's resources or data. A guest VM user is an actor who has access to the virtualized environment and can potentially manipulate the system to launch an attack. Therefore, no component in the driver domain or other guest VMs is considered to be trusted. This means that any potential vulnerabilities in those components are assumed to exist, and appropriate security measures should be put in place to mitigate them. To mitigate threats associated with untrusted components, Kite aims to limit the attacker's flexibility by reducing the attack surface of driver domains (in Xen's driver domain model).

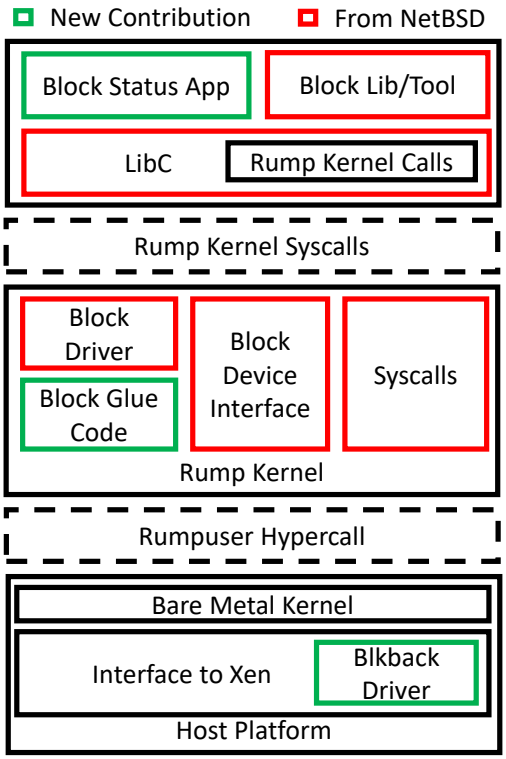


Figure 4.2: Rumprun storage domain.

4.3 Storage Device Driver

A driver domain requires full access to the underlying device that the PV backend driver will use to serve I/O requests from guest VMs. Specifically, a storage driver domain can use a physical storage device, such as a hard disk drive (HDD) or solid-state drive (SSD). The device can be assigned to the driver domain from the Dom0 using Xen's PCI passthrough mechanism, which we particularly use for assigning a storage device to a rumprun unikernel. Once the device is assigned, a corresponding storage device driver is necessary. Therefore, the next step is to get a suitable storage driver that is compatible with the rumprun unikernel.

From the discussion in Chapter 2, we know that this driver belongs to the *rump Kernel* layer in the rumprun stack, as shown in Figure 4.1. Inspired by the rump kernel's philosophy, we aim to import the storage device driver from another OS (i.e., NetBSD) and use it unmodi-

fied. Therefore, it benefits us by saving development and maintenance efforts. However, to realize this vision and ensure that the driver works on the underlying platform and exports its functionalities, we will need to provide the *glue code*. It belongs to the *rump Kernel* layer to bridge the unikernel with the driver obtained from another OS.

Although the storage driver is imported from another OS, it relies on the underlying kernel support from the target OS. The *rump kernel* provides an unmodified system call (*syscall*) interface, which allows the imported driver to utilize *rump Kernel* functionality without any modification. Additionally, to interact with the device driver for storage operations, such as reading or writing on the disk, *block device interfaces* are needed. Like *syscalls*, these interfaces are part of the *rump kernel* layer and can be obtained from other OSs. In the *rump kernel* layer, both *syscalls* and device driver interfaces are referred to as *factions* (as shown in Figure 4.2).

4.4 Storage Backend Driver

In a virtualization setup, multiple guest VMs should be able to use the same or different storage devices. The storage backend driver is the most crucial part of a storage driver domain because it multiplexes the physical storage access between guest VMs and makes fast storage virtualization possible. Each guest machine using PV storage device/s is responsible for providing a blkfront per device. On the other hand, the storage domain is responsible for offering a blkback against each of these blkfronts, as shown in Figure 4.1.

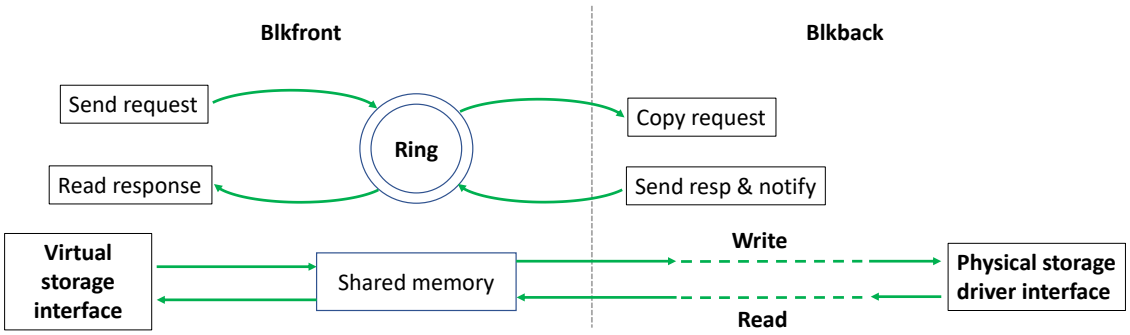


Figure 4.3: Blkfront and blkback communication. For read operations, blkfront places a request in the ring and notifies blkback. Upon receiving the notification, blkback reads data from the storage device, places it in the shared memory buffer, and sends a notification to blkfront so that it can read from the buffer. For write operations, blkfront places the data in the shared memory, places a write request in the ring, and sends a notification to blkback. Upon receiving the notification, blkback writes the data in the storage device and sends a notification to blkfront upon completion.

Our storage domain monitors all guest VMs and their properties all the time to realize when to populate a blkback instance. If any guest VM seeks PV storage, the storage domain negotiates with that VM and creates a blkback instance. Blkback and blkfront communicate and transfer data between each other using Xen's shared memory and event channel mechanisms. Figure 4.3 shows and describes the general flow of communication when a guest VM performs a PV storage device operation. Since blkback driver is very hypervisor-specific, it belongs to rumprun's platform layer, as shown in Figure 4.2.

We divide blkback into platform-dependent and platform-independent layers (upper and bottom) according to the rumprun's philosophy. The bottom layer handles requests for block data from blkfront and sends responses using Xen shared ring buffer. Depending on the type of request (read, write, etc.), the bottom layer communicates with the upper layer. The upper layer performs the read and write operations on the storage device using the block device interface. The upper layer is also responsible for forwarding the feedback from the device driver to the bottom layer.

Each request is consists of multiple block segment information to perform a particular operation, such as read, write, etc., on those segments of the block device. A write-request requires reading block data from the shared memory and writing them to the storage device. In contrast, a read-request requires reading from the storage device and writing them to the shared memory. In Xen, the shared memory access is performed using the grant table operations, which requires hypercall execution, and they are time-consuming.

Blkfront notifies blkback of requests via a Xen event channel. To receive these notifications, blkback has a designated notification handler. To prevent the accumulation of requests and improve request handling, we have implemented a dedicated thread for reading all pending requests, performing the necessary operations on the storage device, and then going to sleep. The notification handler only wakes up this thread when a notification is received. This allows the notification handler to respond promptly when additional requests are received.

Blkback sends a response to blkfront when requested operations are completed on the storage device. However, performing block operations on the storage device can also be time-consuming. If a request is only handled once the response for the previous request is sent to blkfront, it is likely that the number of pending requests will accumulate, causing the ring to become full of requests. This results in high response times, low throughput and high latency, which is not desirable. To address this, the response is sent to blkfront as soon as the corresponding requested operation is completed by the device driver. Instead of waiting for the operation to be completed and then sending the response and handling the next request, we send responses asynchronously. This means we continue handling requests and submitting the requested operations, and only send responses when the storage device has completed those operations. This way, subsequent requests do not have to wait for the response to be sent for the current request.

There are few other optimization techniques we have used in out blkback driver. They are

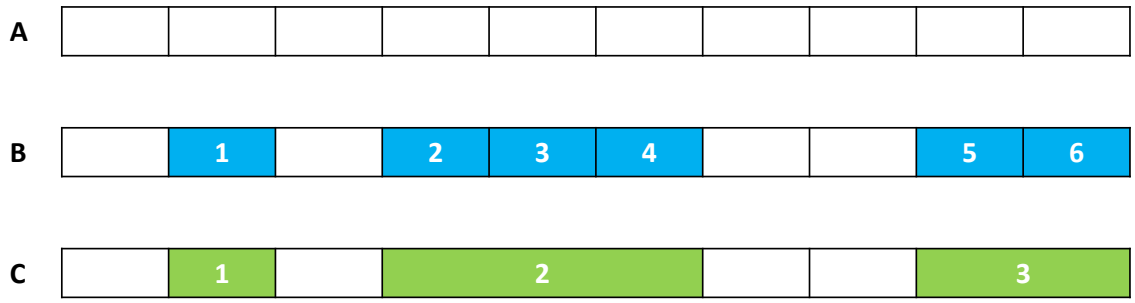
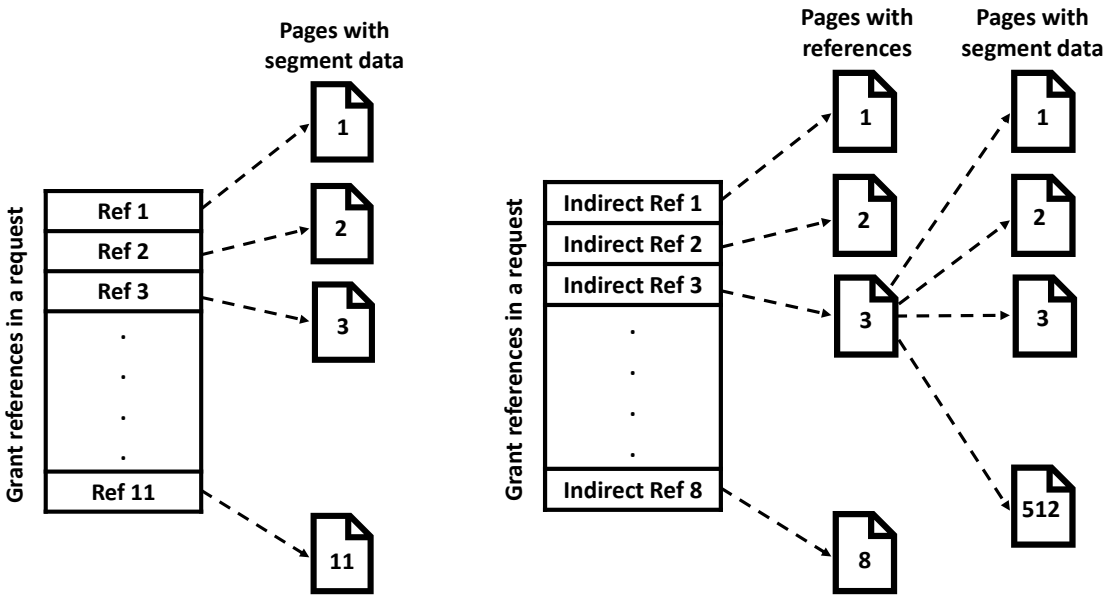


Figure 4.4: Batching multiple segments improves storage domain performance. However, there is no guarantee that segments in a disk (A) will be accessed consecutively. Typically, it would take 6 separate read/write operations to access the segments marked with blue boxes (B). However, using batching, we can perform one operation for segments 2, 3, and 4 combined and another for segments 5 and 6 combined, resulting in a total of 3 operations instead of 6, as shown with the green boxes (C). In this scenario, batching saves half the time required to perform all operations separately.

discussed below.

- **Batching:** A request may contain multiple segments of information to perform operations. It is possible that one segment is consecutive to the next segment. Similarly, multiple requests may have consecutive segments where the first segment of a request starts from the last segment of the previous request. In such cases, if the same operation is to be performed on these segments, we combine these operations into a single operation and perform it on the storage device. Figure 4.4 describes how batching can accelerate storage operations with an example.

- **Persistent reference:** A request may contain multiple grant references, each of which points to a shared memory location. The corresponding segment data is read from or written to these locations by blkfront and blkback. However, the shared memories are allocated by blkfront. Therefore, blkback must map the grant references to access these memories, and unmap them after accessing them. This grant reference mapping and unmapping incurs significant time due to context switching for hypercall. To reduce this cost, we take advantage of Xen's persistent grant referencing feature, which



(a) Direct referencing. (b) Indirect referencing.

Figure 4.5: Both direct **(a)** and indirect **(b)** requests can point to one 4K page per reference, with a maximum of 11 and 8 references per request, respectively. A page referenced by a direct request contains segment data. In contrast, a page referenced by an indirect request can contain up to 512 other references, each pointing to a 4K page containing segment data.

allows blkfront and blkback to reuse already mapped references. This means a grant reference does not need to be unmapped as it can be used again. Later, if any of these references are used for transferring segment data, the corresponding memory can be accessed using previously mapped pages, which significantly reduces processing time.

- **Indirect segments:** One request can contain up to 11 grant references, each of which points to one 4K page of shared memory. Therefore, one request can transfer a maximum of 44K data between blkfront and blkback. This limitation can act as a bottleneck for paravirtualizing high-throughput storage devices, such as NVMe SSDs. Xen supports a feature called indirect referencing to overcome this limitation. Instead of holding block data, indirect grant references point to pages that contain segment information, such as sector numbers and grant references to the corresponding shared memories. This way, each page can hold up to 512 segments of data. Each request can

contain a maximum of 8 indirect segments, making a total of 4096 segment information that each request can transfer. As each of these segments can point to one 4K page of data, each request can transfer up to 16MB of data between blkfront and blkback. In our blkback design, we leverage this feature to improve storage throughput. Figure 4.5 depicts the difference between direct and indirect referencing.

4.5 Storage Domain Application

Xen provides block device scripts for NetBSD and Linux, where these scripts retrieve some storage device information and write them to the Xen store database. This information includes a device's status, grant references for shared buffer rings, event channel number, etc. The storage backend scans through the database to learn about the device. Since we cannot run shell scripts in rumprun, we design an application to retrieve device-specific information and pass them to the backend. Since rumprun is a single address space OS, we schedule context switching, between the backend and this application, in a time-sharing manner.

Chapter 5

Kite's Storage Domain Prototype

The existing rumprun unikernel is not designed for storage domains and lacks backend implementation for I/O devices. Therefore, along with the core PV storage backend driver, the unikernelized storage domain requires other implementations, such as physical device interface, backend invocation, and storage domain application. In this chapter, we present the Kite storage domain, a prototype for the unikernelized driver domain for storage based on rumprun, and the design choices as described in Chapter 4.

In Section 5.1, we discuss how Kite's blkback driver accesses the physical storage driver. In Sections 5.2 and 5.3, we discuss Kite's backend driver invocation, initialization, and its connection setup with the frontend driver. Sections 5.4, 5.5, 5.6, and 5.7 contain blkback implementation detail. Table 5.1 summarizes some new and existing functions that are crucial parts of these implementations. Finally, Section 5.8 describes the storage domain application implementation for complementing Xen's missing storage scripts for unikernels.

5.1 Block Device Interface

In NetBSD, there are several block device interfaces/functions that the storage domain needs to use for performing operations on storage devices. The upper layer of the system is responsible for invoking these functions.

Table 5.1: Key functions in Kite storage domain.

Function	Description
devsw_blk2name	Get block device name
bdevsw_lookup	Get block device structure
xbdw_thread_func	Watch for changes in Xenstore
xbdback_instance_search	Searches for blkback for blkfront
probe_xbdback_device	Probes a blkfront that does not have a blkback
xbdback_xenbus_create	Creates a blkback instance
xbdback_connect	Establish blkback's connection with blkfront
xbdback_frontend_changed	Invoked upon blkfront's status change
xbdback_backend_changed	Invoked upon blkback's status change
xbdback_function	Handles I/O between blkback and blkfront
bdev_strategy	Submit I/O request to physical device driver
xbdback_iodone	Callback function invoked upon I/O completion

Before performing any operation on a device, the first step is to check the device's availability and status. The physical device driver assigns a name to each device. Therefore, as a part of the sanity check, we first check if a device has a name using the `devsw_blk2name()` function. If it returns no name, then the device is assumed to be non-existent.

For each block device, there is a constant data structure called the Block Device Switch (`bdevsw`), in NetBSD. This data structure is defined in the physical driver and includes mandatory entry points such as `open`, `close`, `strategy`, `dump`, `size`, `flags`, `ioctl`, etc. To perform any operation on the device, we heavily depend on the interfaces that allow us to use these entry points. Therefore, the next step is to search for the device's `bdevsw` structure using the `bdevsw_lookup()` function.

In UNIX-based operating systems, a device is treated as a file. Therefore, we create a virtual

node (vnode) for the storage device, which represents an inode of a storage device file. As a result, we can perform `open`, `close`, `read`, and `write` operations on it just like files. In NetBSD, we can obtain a vnode for a block device using the `bdevvp()` function. The next step is to perform the `open` operation on the device, followed by any necessary `read`, `write`, `ioctl`, and `close` operations, which are done using `VOP_OPEN, bdev_strategy(), VOP_IOCTL`, and `vn_close`, respectively.

Reading and writing on a block device is done by first constructing buffers and then submitting these buffers using the `bdev_strategy` function. The buffer serves as a data placeholder and holds various information such as the source address, operation type, and a callback function. Before constructing the buffer, we initialize it with the necessary information and destroy it after we are done using it. We use the device driver interface to initialize and destroy the buffers. The device driver reads data from the buffer and writes it to the storage device for write operations. For read operations, the device driver reads data from the storage device and writes it to the buffer. After a completed operation, the device driver invokes the aforementioned callback function.

The usage of the above-discussed block device interfaces from the upper layer is mostly triggered by the lower layer, except for the callback function. However, the lower layer is the Xen-specific layer executed in the rump kernel context, while the upper layer uses the device interface executed in NetBSD codes. Since rumprun is a single process kernel, we need to switch between the rump kernel and NetBSD code context. Therefore, when we switch from the lower layer to the upper layer, we use cooperative scheduling by exiting from the rump kernel code using the `rumpuser__hyp.hyp_schedule()` function. Later, after upper layer execution, we switch back to the rump kernel context using the `rumpuser__hyp.hyp_unschedule()` function.

```
/local/domain/2/backend = ""
/local/domain/2/backend/vbd = ""
/local/domain/2/backend/vbd/4 = ""
/local/domain/2/backend/vbd/4/51712 = ""
/local/domain/2/backend/vbd/4/51712/mode = "w"
/local/domain/2/backend/vbd/4/51712/state = "4"
/local/domain/2/backend/vbd/4/51712/dev = "xvda"
/local/domain/2/backend/vbd/4/51712/type = "phy"
/local/domain/2/backend/vbd/4/51712/online = "1"
/local/domain/2/backend/vbd/4/51712/bootable = "1"
/local/domain/2/backend/vbd/4/51712/removable = "0"
/local/domain/2/backend/vbd/4/51712/frontend-id = "4"
/local/domain/2/backend/vbd/4/51712/device-type = "disk"
/local/domain/2/backend/vbd/4/51712/discard-enable = "1"
/local/domain/2/backend/vbd/4/51712/params = "/dev/vg/logical_volume"
```

Figure 5.1: Xenstore is a key-value store database that maintains a directory-like structure for keys, each representing a path. This listing is a partial snapshot of Xenstsore database generated using the `xenstore-ls` command. It shows some key-value pairs for the storage driver domain (ID 2), which is connected to a blkfront from guest domain (ID 4).

5.2 Blkback Instantiation

The creation of a guest VM, i.e. DomU, is a discrete event. As a result, a blkfront instantiation can occur at any time, requiring a blkback to be instantiated on the driver domain side to pair and enable PV I/O. To detect blkfront creations, the Kite storage domain creates a dedicated thread to monitor changes to the Xenstore database regarding the storage domain. If a new VM wants to use this storage domain, Xenstore adds a new set of blkfront information to the storage domain's backend path. Figure 5.1 shows a few backend paths in a Xenstore database.

The Kite storage domain maintains a list of important Xenstore paths and sets a callback function for each of them. A dedicated thread runs the `xbdw_thread_func()`, which continuously monitors for changes in the Xenstore database. If there is a change for a path on the list, the corresponding callback function is invoked. Kite uses the `xbdback_instance_search()` function as the callback for any changes in the storage domain's backend path. This function investigates the changes in the path by traversing all blkfront subdirectories. The

Table 5.2: Important storage device information available in Xenstore.

Key	**Description**
frontend	Frontend's path
frontend-id	Frontend's ID
state	Connection state
dev	Device's name
mode	I/O permission
device-type	I/O type
sectors	Number of sectors
sector-size	Each sector size
params	Storage file/disk's path
type	Device type (physical or virtual)
feature-flush-cache	Support for cache flush
physical-sector-size	Sector size for physical device
feature-persistent	Support for persistent referencing
feature-max-indirect-segments	Maximum indirect segments allowed

`probe_xbdback_device()` function detects blkfronts with a `XenbusStateInitialising` status, indicating the creation of a new blkfront instance that is waiting for a blkback to pair.

The `xbdback_xenbus_create()` function first ensures that no blkback instance has been created previously for this blkfront by checking the list of blkback instances. It then creates an instance of blkback for this blkfront. A blkfront instance is an object of the `xbdback_instance` structure, which contains various information about the requested storage device, such as the detail for blkfront, blkback, shared ring, etc. Kite adds the blkfront's and blkback's absolute paths to the watchlist, so that the `xbdback_frontend_changed()` and `xbdback_backend_changed()` methods are invoked, respectively, for any changes in those ends. Lastly, this function reads the device's major-minor (physical-device) number and adds it to blkback path on the Xenstore database.

5.3 Blkback Initialization and Connection

The `xbdback_backend_changed()` function first performs a sanity check on the requested storage device, as described in Section 5.1, requested by the corresponding blkfront. It then adds blkback properties to the Xenstore database, including sector count, sector size, read-/write mode, and information regarding supported features such cache flushing, persistent grant reference, and indirect segments. Finally, it changes blkback connection status to connected (`XenbusStateConnected`). Table 5.2 shows some of the storage device information that is stored in Xenstore.

Upon reading these properties from Xenstore, the blkfront also changes its connection status to `XenbusStateConnected`, which triggers the `xbdback_frontend_changed()` function to be invoked. This causes the `xbdback_connect()` routine to finish the remaining steps to complete the connection between the blkfront and blkback.

In Xen, communication between the frontend and backend parts is achieved through special I/O ring buffers, which are built on top of a shared memory mechanism called the grant table [141]. The ring reference is a reference to the location of the shared memory buffer. The xbdback_connect() function reads the blkfront properties from Xenstore, which include the ring reference, event channel number, support for persistent grant reference, and the I/O protocol. Then, it maps the ring to a page so that blkback can access it, and an event handler is set to handle notifications from the blkfront using the event channel.

5.4 Event Handler and Request Handler Thread

As discussed in Section 4.4, to keep the event handler available for request notifications, a request handler thread (`xbdback_thread`) is created in the `xbdback_connect()` function.

The event handler routine wakes up the `xbdback_thread` if it is not already awake upon receiving a new notification.

The `xbdback_function()` handles the I/O processing using a call graph adapted from NetBSD's blkback implementation. The `xbdback_co_main_loop()` function copies the request into blkback object, and the `xbdback_co_io_loop` keep reading segment information from the copied requests. The `xbdback_co_io_gotfrag2()` function constructs a batch of buffers, which are initialized by `xbdback_co_io_gotio()`, with the segment information. The `xbdback_co_map_io()` function maps the grant references to the pages, from the storage domain's address space, which is used as the block data place holder in the buffer.

The `xbdback_co_do_io()` function invokes the device driver interface using the upper layer to perform I/O operations on the disk. `VOP_IOCTL` interface and `bdev_strategy()` function performs cache flush operations and submits read/write operations, respectively.

5.5 Handling Device Driver Responses

As requests are received from the blkfront, they are processed, and the requested operations are submitted to the physical device driver. However, no response to the requests is sent to the blkfront until the lower layer receives a response from the device driver regarding the submitted operations. The NetBSD buffer structure allows Kite to set a callback function, which is invoked by the device driver when it finishes performing the submitted operations.

Kite storage domain sets a common callback function in the upper layer for all buffers. This function ultimately invokes a function in the lower layer, which can unmap grant references (unless they are persistent), send responses to the blkfront, and destroy used buffers. The `xbdback_iodone()` function sends responses to the blkfront by placing them in the ring,

based on the success or failure of the operations submitted to the device driver. Blkback sends notifications using the event channel to inform the blkfront of the placed responses.

Once all responses to batched operations are prepared and sent to the blkfront, it is important to release all used data structures to efficiently reuse the memory. Therefore, blkback destroys used buffers by invoking the device driver interface and return other used data structures to the pools.

5.6 Persistent Reference

For all segments in one request, it is possible to batch the grant mapping, so that blkback can map the references on consecutive pages. The benefit of doing this is that blkback can use only one buffer to perform operations on the device for all segments in one request. For persistent referencing, Kite does not unmap grant references and tries to reuse previously mapped pages if the corresponding grant references appear in a new request. Otherwise, a grant reference is mapped to a newly allocated page, if not mapped before. However, with persistent referencing, there is no guarantee that the same set of grant references will appear in another request. As a result, blkback cannot reuse the consecutive pages from the batched mapping of those references. Therefore, Kite discretely maps pages for grant references. One mapped page acts as a placeholder for data in a buffer. Consequently, Kite ultimately uses one buffer for each segment. Additionally, Kite batches the buffers to submit all segment operations in one request

To maintain a list of mapped pages, Kite implements a simple array of mapped page addresses. The indices of this array represent the grant references. When a buffer is constructed for a segment, Kite first checks if the corresponding grant reference already has a mapped page in the array. If a mapped page is found, Kite reuses it. Otherwise, it maps the grant

reference to a newly allocated page. On the other hand, before destroying the buffers, Kite returns the pages to the array instead of unmapping them.

5.7 Indirect Segments

The implementation of indirect segments requires extra parsing of requests if it indicates the operation type is indirect. This parsing is done in the `xbdback_co_main_loop()`. First, Kite maps the grant references to pages. Then it parses these pages, each of which may contain up to 512 indirect segments. However, by default Linux supports a maximum of 32 indirect segments. Therefore, we also limit the maximum number of indirect segments to 32.

We store the parsed indirect segments in the `bdi_indirect_segments` in blkback object, so we can read their information in the `xbdback_co_io_loop()` where segments are prepared for I/O operations.

5.8 Application

Each device has a unique major-minor number pair, which is used for identifying devices separately. The Xen block scripts in NetBSD and Linux read this number for the requested devices and add them to the Xenstore database. Since Kite does not have this script, it reads the major-minor number pair for all available devices using a C application. This application constructs a list of devices with their major-minor numbers and availability, indicating if the device is already being used by a guest VM. The list is shared between the PV storage driver and the application.

If the application terminates, the unikernel will halt. Therefore, Kite needs to run the application as long as we want to run the storage domain. To achieve this, once the application

finishes constructing the list, it unschedules itself by yielding periodically, so that the other parts of the unikernel can get CPU context to execute codes.

5.9 Implementation Effort

Table 5.3 shows the gist of Kite's storage domain implementation effort. Most changes pertain to extending HVM support for rumprun and implementing the blkback driver. The application for configuring and retrieving storage information involves very few lines.

Table 5.3: Lines of code (LOC) changed or added for Kite storage domain.

Component	**Description**	**LOC**
Blkback	Xen's storage backend driver	1904
HVM extension	xenbus and xenstore support	1100
Configuration	storage applications	58
Total:		**3062**

Chapter 6

Kite: Unikernelized Network Domain

Network devices are considered essential for guest machines, alongside storage devices. PV network devices utilize shared memory and ring buffers for communication between the backend and frontend. However, networking requires two rings instead of one, as depicted in Figure 6.1. This presents unique challenges that require new design considerations when unikernelizing a network domain. This chapter will cover these challenges, our proposed solutions, and a general overview of our approach.

Section 6.1 examines the design challenges in creating a Kite network driver domain. The association between the physical network driver and PV network backend, as well as the required design changes in the rumprun architecture, are explored in Sections 6.3 and 6.4. The process of linking the physical NIC and PV backends, enabling communication between the guest and the rest of the network, is discussed in Section 6.5.

6.1 Challenges

As discussed in Section 4.1, rumprun lacks I/O backend drivers; specifically, there is no netback implemented, which is an integral part of a network driver domain. Therefore, the biggest obstacle for a unikernelized network domain is to design and implement a netback modifying rumpurn that is compatible with other guest OSs with the network frontend. Additionally, similar to the storage domain in commodity OSs, netback heavily relies on

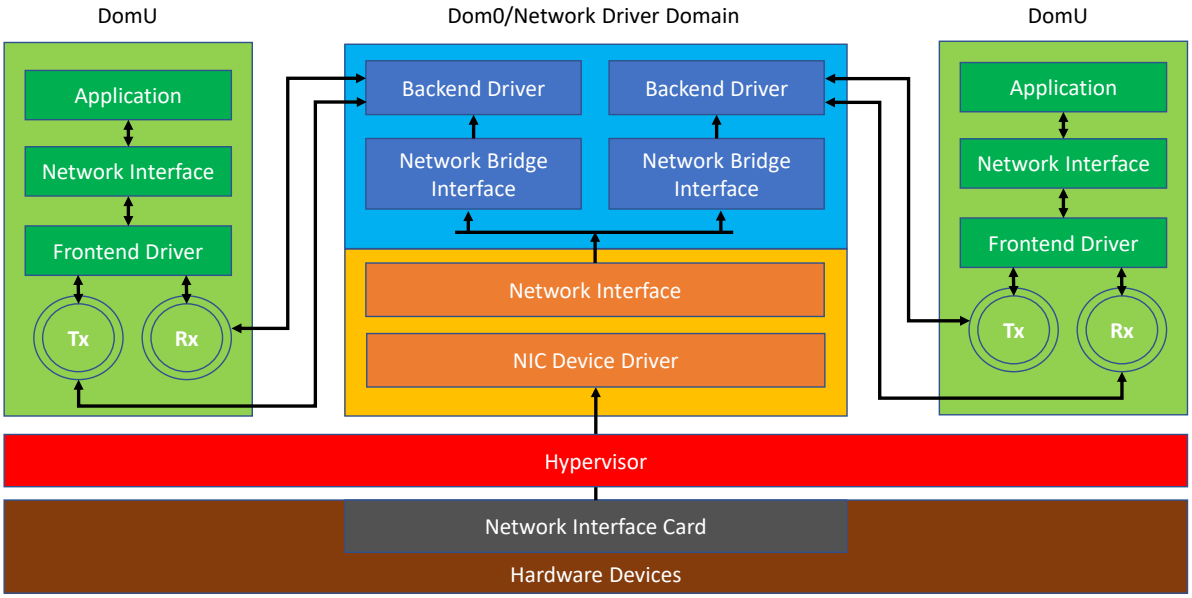


Figure 6.1: Xen's PV network driver model.

various configuration scripts (e.g. network bridging across different backend instances) which cannot be used in rumprun-based environments.

Unikernelizing the network driver domain requires more than porting an existing netback driver from a commodity operating system like NetBSD or Linux. On the one hand, unikernels are single-purpose and single address-space operating systems. Furthermore, a netback driver must consistently manage the data flow for both receive and transmit rings, which take place asynchronously. To ensure smooth receive and transmit operations, netback must communicate with the TCP/IP stack in rumprun and also with the frontend. Therefore, we redesigned netback driver as a single-process unikernel that is capable of seamless duplex communication without sacrificing performance or existing frontend compatibility.

Depending on the number of netfronts in DomUs, there can be several netback instances in the driver domain. Typically, to link these netbacks to a physical NIC, different techniques are used such as bridging, switching, routing, network address translation, and virtual local area networks. These techniques are independent of netback and run as a separate process

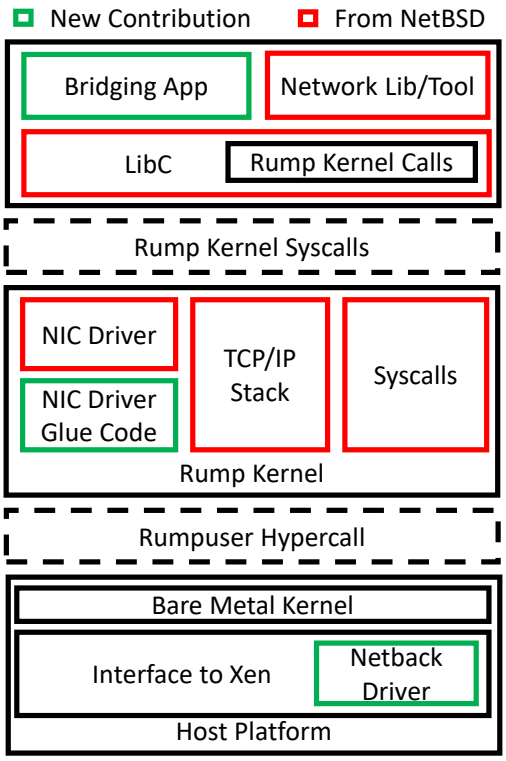


Figure 6.2: Rumprun network driver domain architecture.

that is triggered when necessary. Since rumprun is a single-process application, we cannot run the linking mechanism as a separate process, unlike Linux or NetBSD. Instead, Kite runs a special program in a dedicated thread, so that netback, integrated with the unikernel, links to a physical NIC with minimal performance penalty.

6.2 Threat Model

Although the design and implementation of the Kite network driver domain have additional challenges than that of the unikernelized storage domain, it shares the same threat model as that of the Kite storage domain (discussed in Section 4.2). The trusted components include the Xen hypervisor, Domain-0, and the compiler. Each component in the network domain and guest VMs is untrusted. Therefore, it is crucial to minimize the network driver domain's

attack surface as much as possible.

6.3 Network Device Driver and Interface

Similar to the storage domain, we aim to import physical NIC drivers from an existing OS and use it unmodified to save on development and maintenance efforts for the network domain. And likewise, Kite needs glue code to make it happen and leverage the drivers' functionalities for the underlying netback driver in the rump kernel layer.

To ensure efficient communication between netback and the NIC for exchanging network packets with the outside world, the rump kernel requires several components besides the NIC driver. One of these components is a network interface, which enables the network packet transfer between the two. To facilitate this communication, we use a TCP/IP stack that provides a standardized network interface. The TCP/IP stack, NIC driver, and glue code are considered *factions* in the rump kernel layer, alongside other essential elements such as syscalls, as shown in Figure 6.2.

6.4 Netback Driver

Netback driver plays a crucial role in a network driver domain. For each netfront at each guest VM, the network driver domain should create one corresponding netback instance, as a single VM may require multiple PV network devices and thus multiple netfronts. Netback and netfront communicate and transfer data between each other using the Xen hypervisor's shared memory and event channel mechanisms. To ensure compatibility with the hypervisor, certain parts of netback driver are hypervisor-specific and are thus integrated as part of the host platform, as illustrated in Figure 6.2.

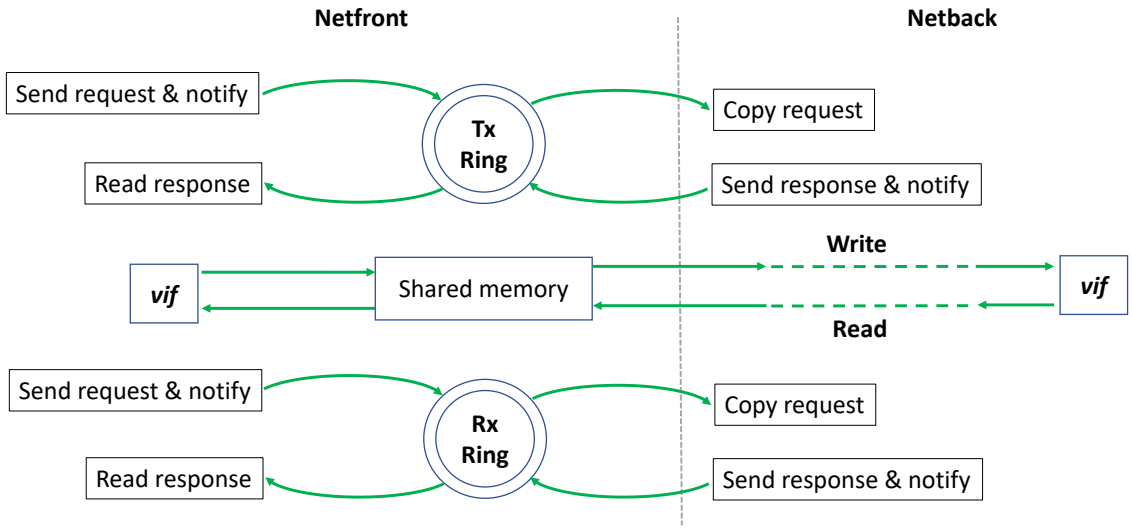


Figure 6.3: Netfront and netback communication. To send packets from DomU, netfront places packets in the shared memory, a transmit request in the Tx ring, and notifies netback. Being notified, netback copies those packets, forwards them to the NIC through netback's *vif*, and notifies netfront upon completing forwarding. A similar sequence of operations is performed using the Rx ring for forwarding network packets from a *vif* to DomU.

Netfront and netback exchange requests and responses using two rings, Tx and Rx, for network packet transmission and reception, respectively. They also use shared memory space for transferring network packets between them. Netfront forwards the guest VM's outgoing network traffic to netback, which eventually forwards it to the physical NIC through netback's virtual network interface (*vif*). On the other hand, netback forwards the incoming traffic at *vif* to the guest VM through netfront. Figure 6.3 depicts the communication between netback and netfront.

The *Interface to Xen* layer hosts netback driver along with other Xen-related implementations such as the xenbus and grand table interfaces. Despite being designed specifically for Xen, we separate netback driver into platform-dependent and platform-independent layers to adhere to the design principles and linking restrictions of rumprun. The upper layer of netback communicates with the NIC driver through rumprun's network stack and linking mechanism, such as a network bridge. For any incoming packet (from the network stack)

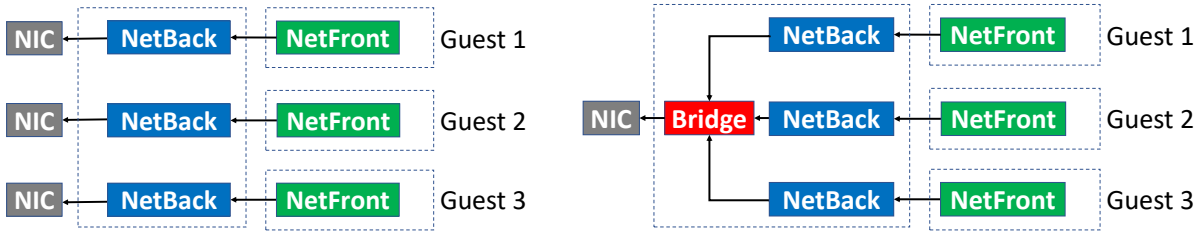
destined for a DomU, this layer places the packets into a memory buffer and forwards them to the bottom layer. The bottom layer then places these memory buffers into the corresponding Xen I/O ring buffer and sends a notification to netfront on the other side.

Similarly, for a stream of network packets originating from a DomU, the bottom layer receives them in memory buffers through the ring buffer and forwards them to the upper layer. Subsequently, the upper layer pushes these memory buffers onto the network stack. Aside from ring buffer operations, other hypervisor-specific operations between netback and netfront such as interrupts through Xen's event channel are done at the bottom layer.

The discussed duplex communication occurs as asynchronous events. Netback is expected to complete the interactions as soon as there is any data available from netfront. However, rumprun lacks rich OS support for interrupts and work queues. In our design, Kite exploits multi-threading so that netback can process data fast avoiding any delay due to hypervisor-based event mechanisms.

Our design includes an event handler that is invoked when there is a notification from netfront for a data request or response. Often, these notifications require operations involving hypercalls, which are time-expensive. Spending significant time in the handler may create a bottleneck, blocking other incoming notifications. We introduce a dedicated thread, activated by the handler, to take over and perform necessary actions in response to notifications.

Similar to netfront notifications, netback needs to respond to the network stack operations. For example, any data received from the stack should be forwarded to netfront. Thus, netback uses network stack callback routines to respond to the network stack operations. Spending too much time in these routines may delay subsequent operations because a response may require hypercall execution, which is expensive. To minimize this response time and prevent the stack operations from blocking, we introduce another thread that is activated



(a) Separate NICs for each netback instance. (b) Multiple netback sharing the same NIC.

Figure 6.4: Linking netbacks can help multiplex a single NIC between multiple guest VMs. Linking methods, like network bridges, can be utilized reduce the need for multiple NICs.

by the routines. This thread performs the processing and execution of the corresponding operations so that the network stack routines can respond to operations without blocking.

6.5 Linking Netback With a Physical Device

To forward network traffic between the outside world and netfront, each netback instance in a network driver domain must be connected to a NIC. However, assigning one NIC per netback, as illustrated in Figure 6.4a, is not a practical solution as a machine can have multiple netback instances. To address this, multiple *vifs* from netbacks are connected to a single NIC.

There are various methods to connect netback *vifs* to a physical NIC. One widely used approach is bridging, which is commonly employed in Xen PV networking. As depicted in Figure 6.4b, a bridge interface can link the NIC interface with multiple *vifs*. How and when bridging should take place is decided by a given driver domain OS. For example, Linux runs a driver domain *daemon* that starts services required in the corresponding Xen driver domain. Among other responsibilities, this service runs networking scripts to establish a bridge.

A single-process unikernel is unable to run multiple services or scripts, thus a *unified application* is introduced. This application creates a network bridge, connects the NIC interface,

and *vifs*. It periodically checks for new *vifs* and adds them to the bridge. This application runs in a dedicated thread, so that it does not interfere with the netback driver. However, if the application runs continuously, it may lead to the netback driver being starved of CPU time. To prevent this from happening, a time-sharing and context switching approach is employed between the netback driver and the bridging application.

Chapter 7

Kite's Network Domain Prototype

The details for Kite's PV network driver domain implementation are discussed in this chapter. A netback instance is needed for each netfront that uses this driver domain for its PV network devices. Netback driver implementation provides virtual network interfaces. The bridge implementation connects these interfaces with the physical network device.

Section 7.1 discusses about the interface between netback and NIC driver. Kite's process of netback creation and its pairing with netfront is discussed in Section 7.2. Sections 7.3, 7.4, and 7.5 talk about the communication between the NIC, netback, and netfront. How Kite use an unmodified NIC device driver from NetBSD [28] for rumprun-based network domain is discussed in Section 7.6. The bridge implementation for connecting NIC and netbacks' *vifs* is discussed in Section 7.7. Finally, Section 7.8 summarizes the implementation effort for prototyping Kite's network driver domain.

7.1 Virtual Network Interface

Each netback instance in Kite network domain creates a *vif* to use as a gateway to communicate with the outside world through the NIC. To do that, Kite leverages the TCP/IP stack from NetBSD, which offers a set of functionalities to talk to the device.

Kite's netback creates the *vif* and sets different callback functions associated with it using the `rump_xennetback_create()` function. The interface is initialized and marked as *running*

Table 7.1: Important network device information available in Xenstore.

Key	Description
backend	Backend's path
backend-id	Backend's ID
frontend	Frontend path
frontend-id	Frontend's ID
handle	Handle ID
state	Connection status
mac	MAC address
bridge	Connecting network bridge
feature-sg	Scatter-gather support
feature-rx-copy	Grant copy support on Rx ring
tx-ring-ref	Tx ring's grant reference
rx-ring-ref	Rx ring's grant reference
event-channel-tx	Tx event channel's grant reference
event-channel-rx	Rx event channel's grant reference
feature-ipv6-csum-offload	Checksum offloading support
feature-split-event-channels	Different event channel for Rx and Tx

by the `if_init()` function. Kite also utilizes the `if_ioctl()` function for any network device-specific `IOCTL` operations. When a packet from the outside world reaches the netback through the interface, the `if_start()` function is invoked. Kite's netback enqueues the outgoing packets to the network stack using the `if_percpuq_enqueue()` function.

7.2 Netback Instantiation and Connection

Xenstore holds information regarding the network devices in guest and driver domains. Table 7.1 lists some PV network device-related crucial keys available in the Xenstore. By

reading these keys, the netback driver can obtain detailed information about the netfront and subsequently publish its own supported features to the Xenstore.

According to the Xen PV driver model, a separate netback instance must be created for every netfront instance in a DomU. Similar to the blkback invocation described in Section 5.2, a netback instance per netfront is created by the network domain. The backend path of the network domain is added to a watchlist for monitoring new netfront instances. A change in that path triggers a corresponding callback function, which investigates the change and instantiates a new netback instance for any new netfront instances.

Section 6.4 states that both netfront and netback drivers utilize two ring buffers, specifically Tx and Rx. Netfront is responsible for initializing the rings and allocating the necessary shared memories. It shares the grant references for these rings and other information, such as event channel references and supported features, with the Xenstore. The netback driver then reads the Tx and Rx ring's grant references and maps them to its own memory address space. Both netfront and netback work together in a producer-consumer fashion to exchange requests and responses using the ring buffers.

The communication between netback and netfront happens asynchronously. Therefore, event channels (virtual interrupts) are used for asynchronous notifications to communicate between front- and back- ends effectively in Xen. Netback binds the event channel to the event handler so that relevant notifications can be handled. For both transmit and receive, a network PV driver can use the same event channel. However, Xen supports separate event channels for transmit and receive as well. Our netback implementation supports both types.

Since hypervisor has all machine memory mapped for all domains, it is faster to copy data from one domain to another using hypervisor. To provide faster data transfer between netfront and netback, Xen supports hypervisor-based data copy, and nowadays, most of the

netfronts from OSs such as Linux and NetBSD utilize this advantage. Therefore, in our Kite network driver domain, we implement this feature.

To distinguish one netback *vif* from another in the network driver domain, each *vif* is assigned a unique name. The first step for initializing a netback instance is to update the Xenstore database so that this netback's properties, such as *vif* name, event channel type, support for data copy, etc., are advertised to other domains.

7.3 Transmit

In a DomU, any data pushed to a corresponding interface is received by the netfront. Netfront transmits the network data to the netback using the Tx ring buffer. Netback forwards the received data through its *vif*. The ring buffer consists of multiple slots that can be used for requests as well as responses. A Tx request means a request to transmit data from the netfront through netback. Each request consists of the address of the page containing data, corresponding offset, relevant flags, size of data, and ID of the request. As shown in Figure 7.1, after inserting request(s) in the ring buffer (T3), netfront sends a notification via an event channel. The notification handler in netback copies the unhandled request(s) (T4) and maps the pages in memory using the grant table mechanism (T5). The memory contents are then delivered to the *vif*s (T6).

Finally, netback sends responses to netfront on the accomplished Tx operations by utilizing the slots from already served requests. Once these slots are filled with responses, netback pushes them to the Tx ring buffer (T7) and updates netfront by sending an event notification if required. During the request serving process, netback updates its Tx request's consumer and response producer indices.

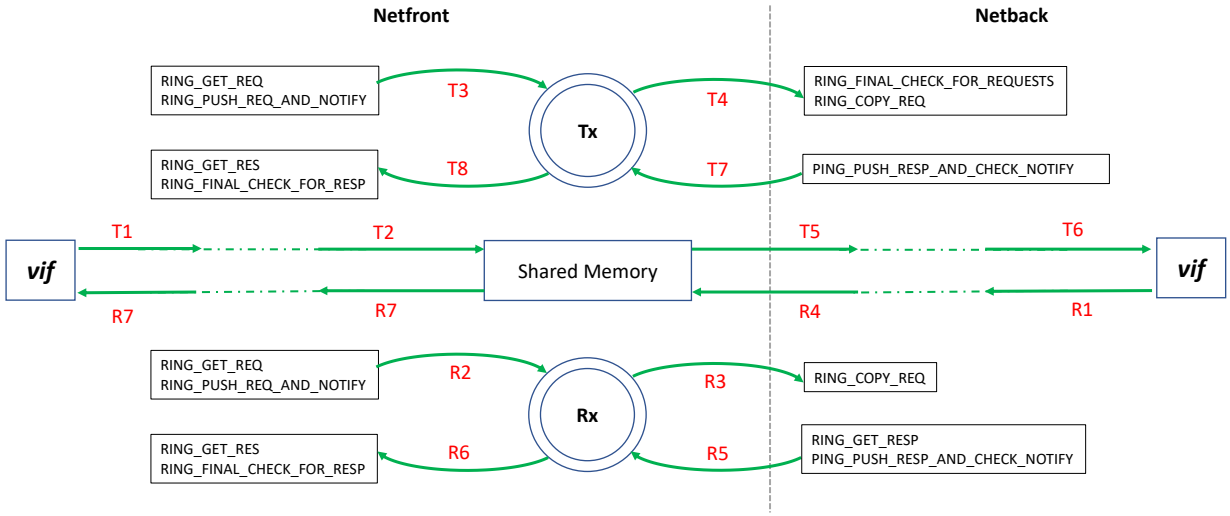


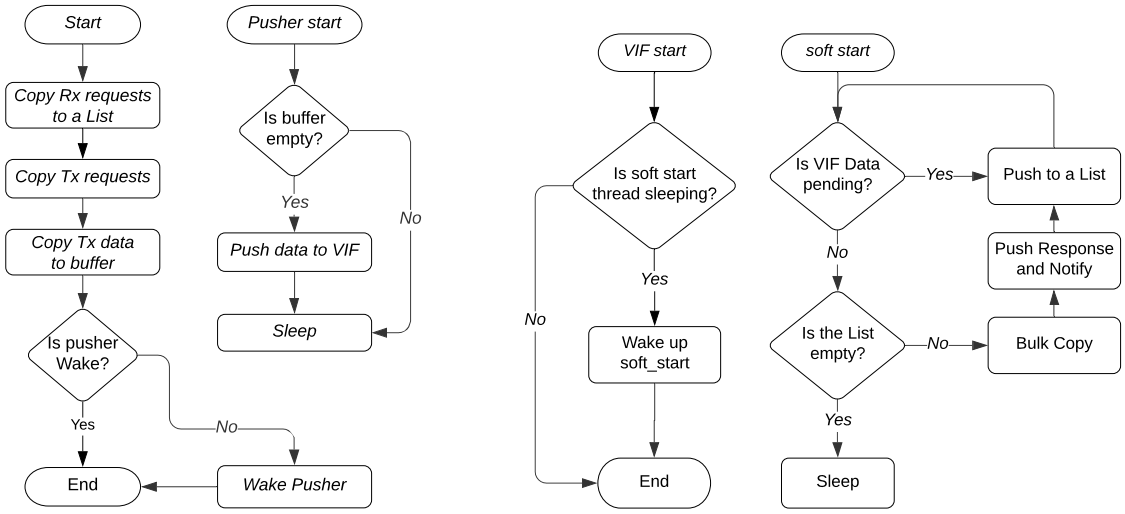
Figure 7.1: Data exchange between netback and netfront.

7.4 Receive

Upon receiving network data from netback, netfront delivers them to its associated *vif*. The role of netback is to receive the data from its attached *vif* and push it using the Rx ring buffer. Like Tx, the Rx ring buffer consists of multiple slots that can be used for requests and responses. However, the front and back end interaction through the Rx ring is not inverse of the Tx ring but similar in some aspects.

Upon receiving network data, netback delivers it to the associated *vif*. The role of netback is to receive data from the *vif* and push it using the Rx ring buffer. The Rx ring buffer, like the Tx ring buffer, consists of multiple slots for requests and responses. However, the interaction between the front- and back- end through the Rx ring is not the inverse of the Tx ring, but it is similar in some aspects.

When netfront sends an Rx request (R2), netback copies the request (R3) but waits until it receives data from its associated *vif*). A function in netback is invoked when its *vif* receives network data (R1), indicating that there are pending incoming data. Netback then copies



(a) Notification handler and pusher thread. (b) Handling incoming data from *vif*.

Figure 7.2: Threaded implementation of netback for efficient interaction with netfront.

the data (R4) into the pages associated with the copied Rx requests using the grant table mechanism. To inform netfront of the copy operation's outcome, netback reuses the served request slots for responses. After pushing the responses to the Rx ring (R5), netback notifies netfront if necessary. Like the transmit operation, netback updates all indices of requests and responses for the Rx ring (R6).

7.5 Threaded Implementation

Rumprun is a single-process unikernel and lacks any rich support for work queues. Since network packet reception and transmission are independent of each other, Kite uses multi-threading to improve the speed of Tx and Rx operations.

Netback's event handler is invoked when it receives a notification. Netback must not block the handler for a long so that a notification can be received and handled as soon as possible. However, notification handling requires the manipulation of shared pages. It involves Xen hypercalls, which are expensive and thus inappropriate for latency-sensitive contexts. There-

fore, Kite uses a separate thread called *pusher*, dedicated to push data to the corresponding netfront instance using hypercalls.

As shown in Figure 7.2a, the handler copies pending Rx requests from netfront to a list. Next, it reads Tx requests and copies the corresponding data to a buffer. Unless the pusher thread is already awake, the handler then wakes up the pusher thread so that the thread can push the buffer contents to the *vif*. Therefore, the handler does not have to wait for the hypervisor to finish copying data to complete its routine. The pusher thread goes to sleep if there is no pending data on the buffer.

On the other hand, when any data is available at the *vif*, a callback function called *start* is invoked. Here, netback's responsibility is to send this data to netfront using Xen's grant table copy mechanism if there is any pending Rx request, which is time-consuming. Thus, if the start function does the copy operation, consecutive incoming data at *vif* may have to wait until the copying is done. Moreover, netback may have to perform multiple copy operations, which could be accomplished with one copy. Therefore, Kite introduces another thread called `soft_start`, which performs copy operations in a batch using only one hypercall. As shown in Figure 7.2b, the start function's only responsibility is to wake up the `soft_start` thread. This thread reads the Rx request from the list, performs the copy operation, and goes to sleep if the list is empty. As a result, received data at the *vif* is forwarded as soon as possible. Table 7.2 lists some crucial functions and their role in Kite's network domain.

7.6 Physical Network Device Driver

The design of rump kernels allows us to utilize existing NetBSD network drivers without modification. By using the PCI passthrough mechanism (with IOMMU for added protection and security), Xen assigns the corresponding PCI device (NIC) to the Kite network driver

Table 7.2: Key functions in Kite network domain.

Function	**Description**
xennetback_entry	Entry point for network backend driver
xennetback_probe_device	Probes a netfront that does not have a netback
xennetback_xenbus_create	Creates netback instance with *vif*
xennetback_connect	Establishes connection with frontend
xennetback_init_watches	Sets callback functions on Xenstore path changes
xennetback_frontend_changed	Invoked for changes in frontend path
xennetback_ifstart	Invoked on data at *vif*. Wakes up *soft_start* thread
soft_start_thread_func	Gets data from *vif*
xennetback_copy	Pushes data to netfront
xennetback_evthandler	Wakes up the *handler thread*
xennetback_network_tx	Gets data from netfront
if_percpuq_enqueue	Pushes data to *vif*

domain. We leverage the 1Gbit and 10Gbit ethernet drivers from NetBSD for the Kite network domain. In addition to the NIC driver itself, Kite uses the TCP/IP stack and other network stack components sourced from the rump kernels in NetBSD.

7.7 Bridging Application

Using a network bridge, Kite connects the *vif* interfaces from netbacks to the physical NIC. To that end, we developed a bridging application in our Kite network driver domain. When this application is launched, it creates a bridge interface. Next, the application assigns an IP address to the physical interface, which works as a gateway for incoming and outgoing packets across all *vif*s. Then, the application keeps scanning for new network interface creations. When a new *vif* appears, the application adds the new interface to the bridge.

We ported the *ifconfig* utility from NetBSD to initialize bridge interfaces and assign IP addresses. Along with that, we also ported the *brconfig* utility from NetBSD, which is used for adding interfaces to a bridge. In order to give CPU context to the other factions such as netback driver, physical driver, and network stack, the bridging application yields its schedule in a time-shared manner.

7.8 Implementation Effort

Table 7.3 summarizes Kite's network domain implementation effort. Most changes belong to rumprun's extended HVM support and the netback driver implementation. In contrast, bridge-related changes are relatively small, making it easier to adapt our driver domain for other network configurations. We also source ifconfig and brconfig utilities from NetBSD.

Table 7.3: LOC changed or added for Kite network domain.

Component	**Description**	**LOC**
Netback	Xen's network backend driver	2687
HVM extension	Xenbus and xenstore support	1100
Configuration	unified (bridging) application	369
Utilities	ifconfig/brconfig changes	222
Total		**4378**

Chapter 8

Kite's Storage Domain Evaluation

For the storage domain performance evaluation, we have built Kite exclusively for storage domain. This chapter discusses the evaluation results, which help to determine if the storage domain implementation incurs any performance overhead compared to existing counterpart solutions. Since all existing driver domains use Linux and there are no existing NetBSD-based driver domains, we only compare against a Linux-based storage driver domain.

Section 8.1 explains our experimental setup for the Kite storage domain's performance evaluation. Section 8.2 examines the read and write performance of a guest VM that uses Kite and Linux domains for PV storage devices. Sections 8.3 and 8.4 discusses similar guest VM's performance for real-life application such as database server, file server, and web server that involves storage operations.

8.1 Experimental Setup

This evaluation uses both micro- and macro-benchmarks. We do microbenchmarking using dd [15] to measure overall storage device throughput. We use macrobenchmarking tools such as Sysbench [125] and Filebench [126] to measure the performance of real-life applications like MySQL [20] and MongoDB [19] database server, file server, web server, etc. We install the Ext4 filesystem on the virtual storage device in DomU to run macrobenchmarks. For each run, we flush the read buffer and use a total I/O size larger than the main memory,

Table 8.2: Configuration of Xen domains on the server side.

	Dom0	**DomU**	**Linux Blkdom**	**Kite Blkdom**
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Rumprun-SMP [82, 103]
Kernel	5.3.0-40-generic	4.15.0-88-generic	5.0.0-23-generic	Based on NetBSD 8.0
Memory	8 GB	5 GB	2 GB	1 GB
vCPUs	1	4	1	1

letting us exercise storage domains more aggressively.

Table 8.1: Hardware configuration.

CPU	1 x Intel(R) Xeon(R) CPU E5-2695, 2.20GHz
Cores	24 per CPU (HT)
L1/L2 cache	32 KB / 256 KB per core
L3 cache	30720 KB
Main Memory	64 GB
Storage	Samsung 970 EVO Plus 500GB NVMe SSD

Table 8.1 shows the hardware configuration for the machine that we use for setting up Xen 4.9 and running the Dom0, storage driver domains, and the guest domain (DomU). The configurations for these domains are shown in table 8.2. The NVMe SSD is assigned to the storage driver domain using the PCI passthrough. The mentioned applications run on the DomU and access the NVMe SSD using the Linux storage frontend. VM configurations are kept similar to get a fair performance comparison.

For most of the cases, Kite's storage domain performs as good as Linux. Certain Kite's storage performance gains can be attributed to NetBSD itself. Other gains are due to the elimination of kernel layers and user space.

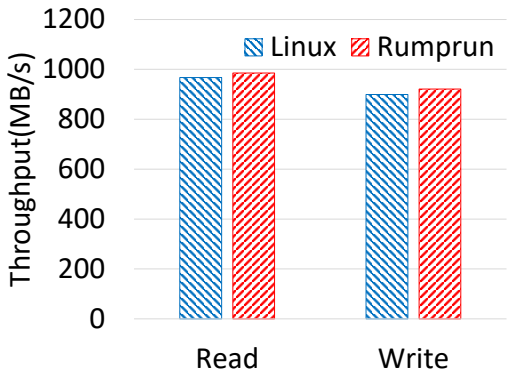


Figure 8.1: Throughput measurements using dd.

8.2 dd

dd is a command line tool in Linux, which lets us to perform read and write operations directly on the storage device. Therefore, we use this tool in DomU to measure and compare read-write performances on the PV storage device while using Linux and Kite for storage domain. To keep the reading overhead minimal during write performance measurement for the disk, we use `/dev/zero` as the source device for write content. On the other hand, to keep the writing overhead minimal while measuring read performance, we use the `/dev/null` as the destination disk. Each experiments are run three times and each time 10GB of data is read-write from/to the device.

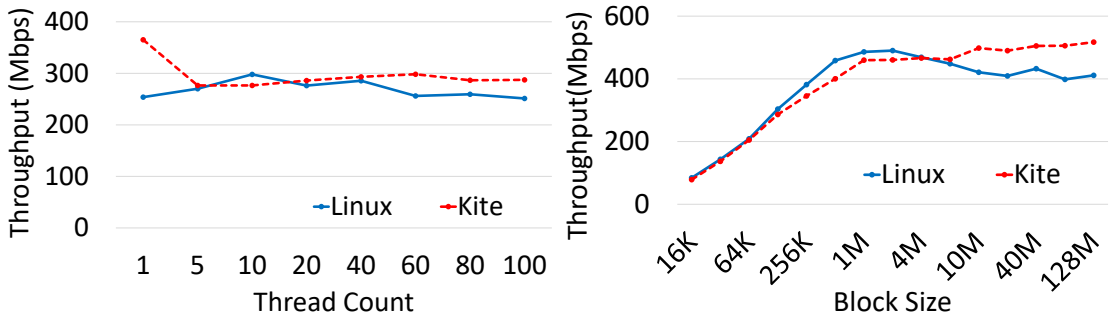
The experimental results are depicted in Figure 8.1, showing the storage device access throughput in MB per second. As one can see, for both read and write operations, Linux and Kite storage domain exhibits similar performance, meaning our unikernelization techniques do not incur any overhead. The relative standard deviation (RSD) for these experiments are 0.062% and 0.038% for Linux and Kite storage domains, respectively.

8.3 SysBench

SysBench [125] is a popular system component benchmarking tool, which is capable of running requests in threads so that multiple requests can run in parallel. For storage evaluation, we leverage SysBench (v1.1.0) for getting ideas on real-life application performance, running them on DomU that is using Linux and Kite storage domain.

8.3.1 SysBench File I/O

We measure the file I/O performance using the SysBench benchmark. For this experiment, SysBench uses 192 files totaling 15GB and performs random operations on these files with a read-write ratio of 3:2 since read operations are performed more than write, in general. We run the same experiment for a different number of threads, ranging from 1 to 100, and block sizes, ranging from 16KB to 128MB. Each experiment is run for 5 minutes, and we make sure to clear the read buffer cache between each run so that we can get actual storage throughput, not primary memory throughput.



(a) Throughput varying number of threads. (b) Throughput for different block sizes.

Figure 8.2: File I/O Throughput measurements using sysbench.

Figure 8.2a shows throughputs for runs with a different number of threads performing I/O operations with each block of size 256KB. On the other hand, Figure 8.2b shows throughputs

for a fixed number of threads (20) but runs with a varying number of block sizes. As evident from the figures, the throughput for Kite storage domain is very comparable with Linux and even better than Linux for higher number of threads and block sizes. The average RSD is 0.49% and 0.33% and the average latency is 16.91ms and 15.23ms for Linux and Kite storage driver domains, respectively.

8.3.2 SysBench MySQL

We setup the widely used MySQL database on the storage device in DomU and evaluate the database performance using SysBench. This database contains 100 tables, each with 1,000,000 records, which totals 20GB of disk space. Since the primary memory size for DomU is 5GB, full database cannot fit in there, which reduces read buffering effect. We ran the benchmark varying number of threads (from 5 to 100) for performing complex SQL queries on the database. Each experiment was run for five minutes.

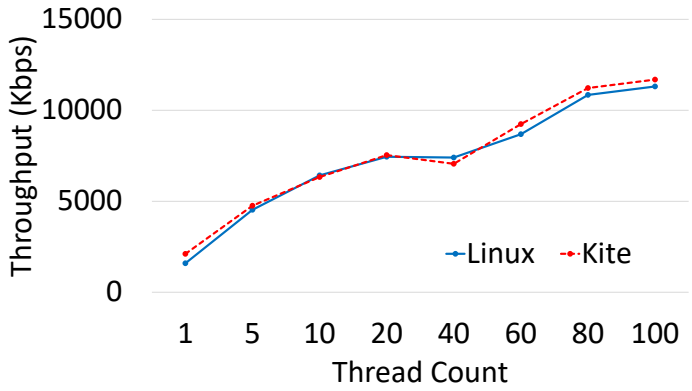


Figure 8.3: MySQL throughput measurements using SysBench.

Figure 8.3 presents the SysBench throughputs for the database operations for a different number of threads. It shows both Linux and Kite have overlapping performances for running the storage domain. It suggests that unikernelizing driver domains with the presented optimizations can help achieve state-of-the-art performance. The maximum average RSD

for these experiments is 0.61% and 0.67%, and the average latency is 18.39ms and 17.83ms for Linux and Kite storage driver domains, respectively.

8.4 Filebench

Filebench is a tool suite that is heavily used, by researchers in academia and industry, for benchmarking works associated with filesystem and storage. It is shipped with several macrobenchmarks written in workload modeling language (WML) and capable of creating workloads of real-life applications. We took a few of them and modified them to benchmark the storage domain that virtualizes high-speed NVMe SSD.

8.4.1 Filebench File server

To generate a file server workload, we ran 50 threads in parallel, each performing a series of operations like create, read, write, append, close, stat, delete, etc. Before running the workload, Filebench creates 100000 files with an average size of 128KB, which makes around 13GB, i.e. at least twice the bigger than the assigned primary memory. The mean append size is 1KB, where I/O sizes are varied, from 16KB to 8MB, for each run for 5 minutes.

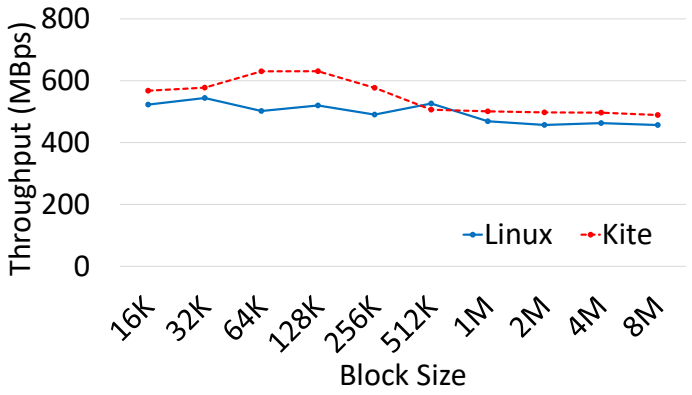


Figure 8.4: Fileserver throughput measurements using filebench.

The throughput for the described file server workload is presented against a different number of I/O block sizes in Figure 8.4. As one can see, the Kite's storage domain often performs slightly better than Linux. We attribute this performance gain is due to the absence of context switching regarding system calls. The maximum incurred latency for this experiment is 8.99ms and 7.93ms and maximum RSD is 0.831% and 0.245% for Linux and Kite storage domains, respectively.

8.4.2 Filebench MongoDB Server

We also evaluate MongoDB, a NoSQL database server, using Filebench because of the difference in its file access patterns compared to the other types of databases. We create 20GB of data with a mean I/O size of 4MB. Figure 8.5 shows the throughput, execution time per operation, and latency, stretched in a logarithmic scale, for a run of 5 minutes with one user. Kite outperforms Linux, proving our storage domain can exhibit better performance even for lower concurrency. This slight performance gain is achieved from Kite's lightweight design, which involves no address space separation. The RSD for this experiment is 0.58% and 0.91% for Kite and Linux storage domains, respectively.

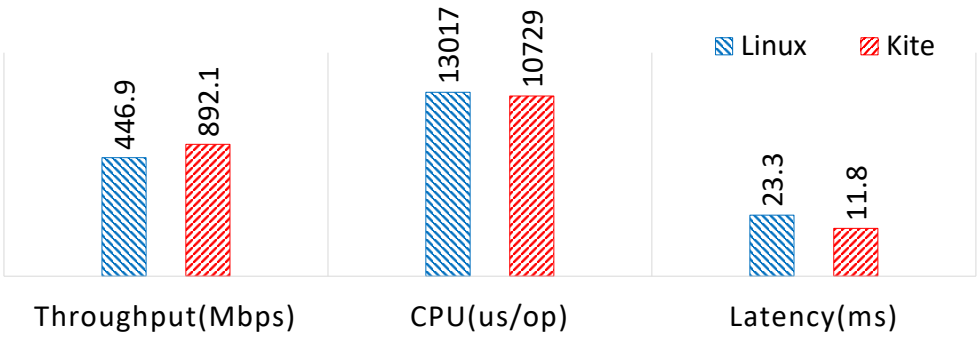


Figure 8.5: MogoDB server performance measurement using filebench.

8.4.3 Filebench Web server

We generate the web server workload by running 50 threads in parallel, each performing a series of operations that combine open, read, and close. First, Filebench creates 200,000 files with an average size of 64KB, totaling around 13GB, to make it at least twice as big as the assigned primary memory. Therefore, the effect of the read buffer is mitigated. The mean append size is 16KB and the I/O size is 1MB, with each operation being run for 5 minutes.

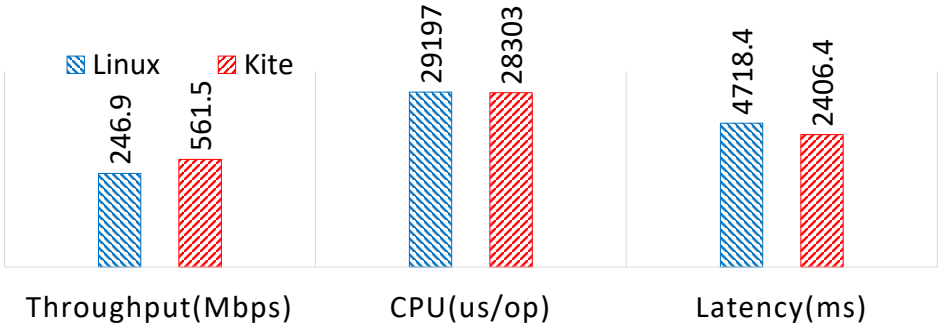


Figure 8.6: Web server performance measurement using filebench.

Figure 8.6 shows the web server throughput, execution time per operation, and latency. It demonstrates that the Kite storage domain takes slightly less time than Linux to execute each operation, providing slightly better throughput. At the same time, the Kite storage domain exhibits faster latency than its Linux counterpart. We attribute this slight performance difference to Kite's single address space architecture, which involves no context switching due to system calls. The RSD for this experiment is 0.94% and 0.71% for Kite and Linux storage domains, respectively.

Chapter 9

Kite's Network Domain Evaluation

In this chapter, we evaluate Kite, which is only built for the network driver domain, discarding all unrelated components. The primary goal of our evaluation is to determine performance overheads, if there are any, because of our implementation. Like the storage domain evaluation in Chapter 8, we only compare against a Linux-based network driver domain. We have confirmed that our system shows similar performance trends for 1Gbe and 10Gbe (with various NIC drivers). In general, Kite should work with any NIC drivers from NetBSD, including recent 40Gbe drivers, for which we expect similar performance trends.

Section 9.1 discusses our experimental setup for evaluating Kite's network driver domain prototype. Guest VM's network bandwidth and latency, when using Kite and Linux network domains, are discussed in Sections 9.2 and 9.3. Finally, Sections 9.4, 9.5, 9.6 evaluate this prototype with several popular real-life applications.

9.1 Experimental Setup

Table 9.1 shows our experimental setup. The client and server machines are directly connected by an SFI/SFP+ network cable, where each has a 10Gbe NIC attached. Driver VMs are tested on the server side. For network-related tests, our client acts as a load generator. The server runs Xen (Dom0 is only used for the control path). Each tested application runs inside DomU. Both Linux and Kite network driver domains have direct PCI passthrough

access to the NIC. Kite environment also incorporates recent changes from LibrettOS [103] to run in Xen's HVM mode.

Table 9.2 shows the Kite network domain's configuration and software versions. We use out-of-the-box Xen 4.9 from Ubuntu. Driver domains are I/O-intensive and do not need to allocate more than one vCPU in our experiments.

Table 9.1: Hardware configuration.

	Server	**Client**
CPU	Intel(R) Xeon(R) CPU E5-2695, 2.20GHz	Intel(R) Core(TM) i5-6600K 3.50GHz
Cores	24 per CPU (HT)	4 per CPU
L1/L2 cache	32 KB / 256 KB per core	32 KB / 256 KB per core
L3 cache	30720 KB	6114 KB
Main Memory	64 GB	16 GB
Network	Intel 82599ES 10-Gigabit adapter	Intel 82599ES 10-Gigabit adapter

Table 9.2: Configuration of Xen domains on the server side.

	Dom0	**DomU**	**Linux Netdom**	**Kite Netdom**
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Rumprun-SMP [82, 103]
Kernel	5.3.0-40-generic	4.15.0-88-generic	5.0.0-23-generic	Based on NetBSD 8.0
Memory	8 GB	8 GB	2 GB	1 GB
vCPUs	1	22	1	1

For the Kite network domain evaluation, we use both micro- and macro-benchmarks. We run the nuttcp [31] microbenchmark to measure overall network throughput. We measure

network latency using ping, Netperf [21], and memtier [22] benchmark. We use macrobenchmarks like ApacheBench [127], Redis [23], and MySQL [20] to measure the performance of real-life applications, which can be relevant to cloud users.

9.2 Nuttcp

We measure the network throughput of a guest VM running Linux that uses 10Gbe NIC through the Linux and Kite driver domains. To achieve optimal throughput with minimal packet loss, we run nuttcp benchmark [31] (v8.2.2) in the UDP mode with 4MB of window size and 8KB of buffer size. As shown in Figure 9.1, with the described configuration, the nuttcp client achieves about 7Gbps with less than 1.5% UDP packet loss for both Linux and Kite network domains.

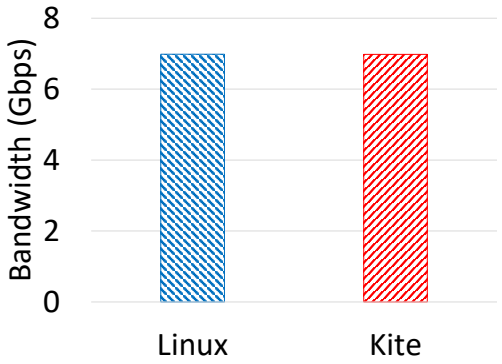


Figure 9.1: Nuttcp throughput for UDP file transfers.

We repeat each experiment 3 times and report the average, where the RSD is 1.33% and 1.02% for Linux and Kite network domains, respectively. The result suggests that Kite exhibits similar UDP throughput to Linux, incurring no penalty.

9.3 Latency

We use various tools with different configurations for measuring network latency. Figure 9.2 shows the latency comparison when using Linux and Kite network backend. Pinging the guest machine from the client machine 100 times with a one-second interval, we get lower latency for Kite (0.31ms) than for Linux (0.51ms). The Netperf [21] benchmark, which sends 1000 requests per second with even intervals to the guest machine, shows 0.18ms latency for Linux and 0.10ms latency for the Kite network domain. Memtier [22], a benchmark for Memcached [18], reports 0.16ms and 0.15ms for Linux and Kite, respectively, when performing 100000 SET and GET operations with a ratio of 1:10 and data of size 8KB.

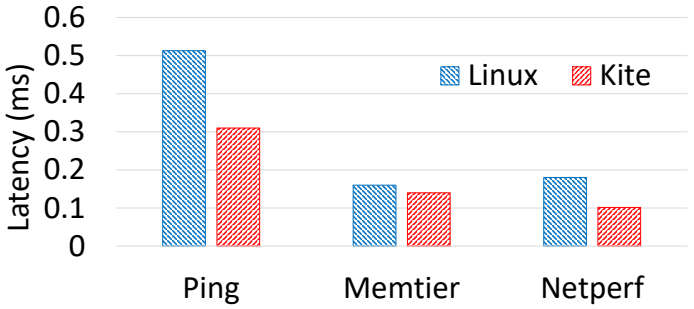


Figure 9.2: Latency comparison for Linux and rumprun network driver domains.

One can see that the Kite network domain achieves slightly better latency than that of Linux across different applications. Therefore, using Kite driver domains for running latency-sensitive applications, we can achieve a similar performance to that of Linux.

9.4 Apache

To evaluate performance with a real-life HTTP server, we run the Apache server (v2.4.29) in DomU and Apache benchmark in the client machine. The server data (files) are randomly generated. The benchmarking tool sends 100,000 requests and measures the server-side

throughput, and each experiment is repeated 3 times. Figure 9.3 shows Apache server throughput for different file sizes, ranging from 512B to 1MB, with 40 concurrent requests for each run. These numbers show the Apache server does not incur any overhead due to Kite's implementation.

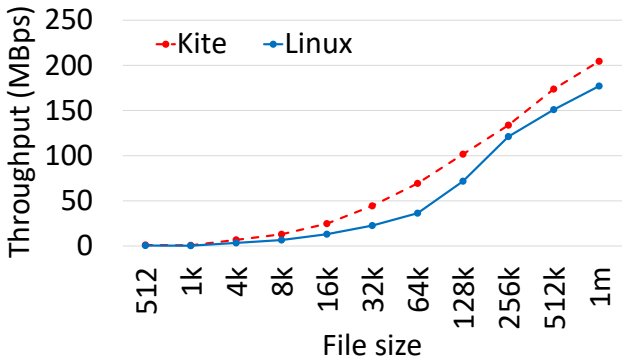


Figure 9.3: Apache throughput varying file size.

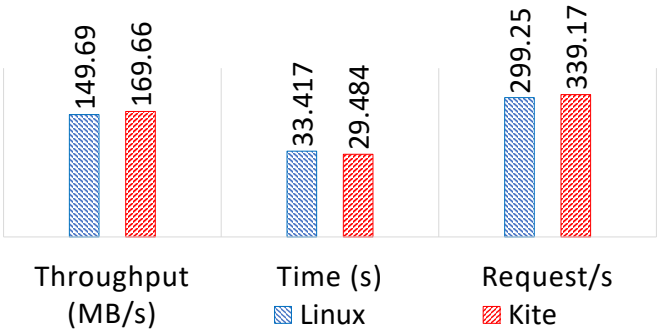


Figure 9.4: Throughput, transfer time, and request rate.

We show a specific example with transfer time, throughput, and request handling rate for 64KB in Figure 9.4. The Apache benchmark sends 100,000 requests with 40 concurrent requests. The performance is marginally higher for Kite. The maximum RSD is 1.20% and 1.44% for Linux and Kite, respectively, for the presented throughputs.

9.5 Redis

Nowadays, in-memory key-value databases are extensively used in cloud infrastructures. We run Redis [23] server, a well-known key-value store, to compare Kite and Linux network driver domains. We execute the Redis benchmark (v4.0.9) with millions of SET/GET operations in the pipeline mode. We set the pipeline size to 1000. Each run is repeated for different levels of concurrency, wherein each GET/SET operation reads/writes 128KB of data with each key of the size of 64 bits.

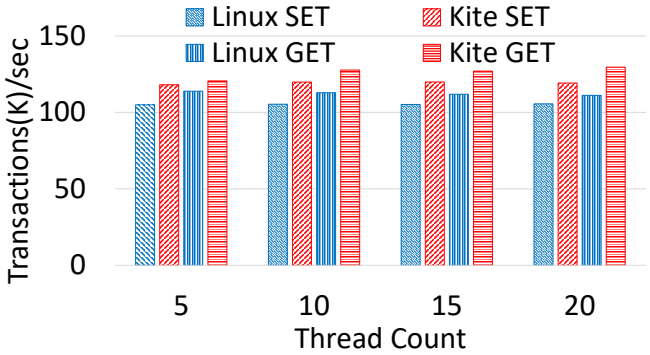


Figure 9.5: Redis key-value store throughput.

Figure 9.5 shows the number of SET/GET operations per second. Overall, Kite and Linux network driver domains exhibit similar performance. Each experiment was repeated three times and the RSD for Linux's and Kite's netback is 0.00053% and 0.0011%, respectively.

9.6 MySQL

Along with NoSQL databases like Redis and Memcached, relational databases are also widely used. We compared the performance of MySQL [20] server (v5.7.29), a popular SQL database, running on DomU. On the client machine, we ran Sysbench (v1.1.0) benchmark, which triggers database transactions sending SQL queries, to measure the database throughput.

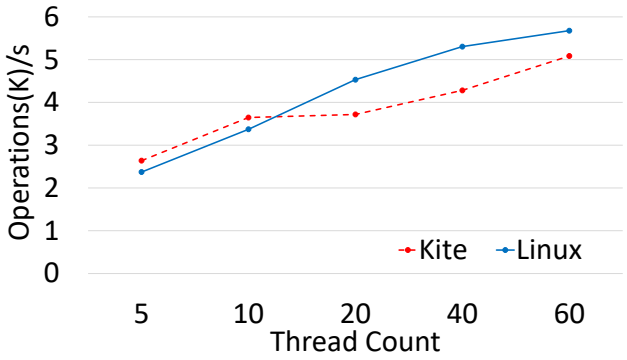


Figure 9.6: MySQL throughput.

We created a database with ten tables, each with 1,000,000 records. All data fits in memory, i.e., the workload is memory-bound, and there is no storage I/O. We ran the benchmark from the client machine for a different number of threads (from 5 to 60). This benchmark sends read-only SQL queries to the server, which allows us to stress-test the network path. Figure 9.6 shows the number of operations (queries and transactions). There is almost no performance difference when using Linux's or Kite's netback. We repeated each experiment three times and the RSD is 0.0167% and 0.0496% for Linux and Kite, respectively.

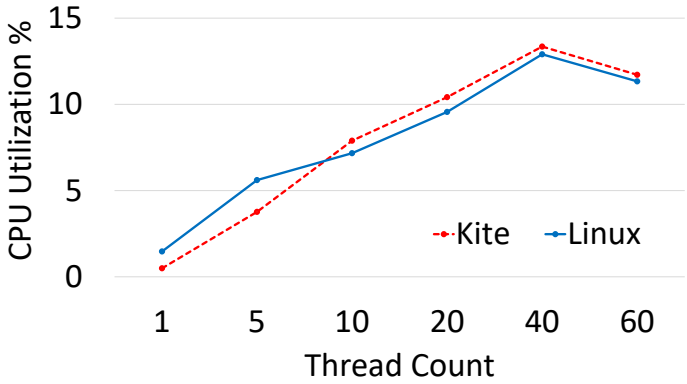


Figure 9.7: CPU utilization while benchmarking the MySQL.

We also measured CPU utilization. Figure 9.7 shows the average CPU utilization of DomU, measured using the sysstat utility [27], during the mentioned benchmark execution. We observed that DomU's CPU utilization for both Linux and Kite is very similar. Therefore, the CPU utilization time is preserved.

Chapter 10

Kite's Security Evaluation

Device drivers are one of the most erroneous parts of any OS, and general-purpose full-featured OSs have significantly bigger attack surfaces. The biggest motivation for unikernelizing Xen driver domains is reducing attack surface so that such critical VMs becomes more resilient against the attacks due to exploitation of presented bugs and, therefore, the cloud infrastructure can enjoy more security. In this chapter, we conduct a thorough security evaluation of Kite's driver domain implementations, including an analysis of the resulting attack surface and a discussion on its strength against related CVEs.

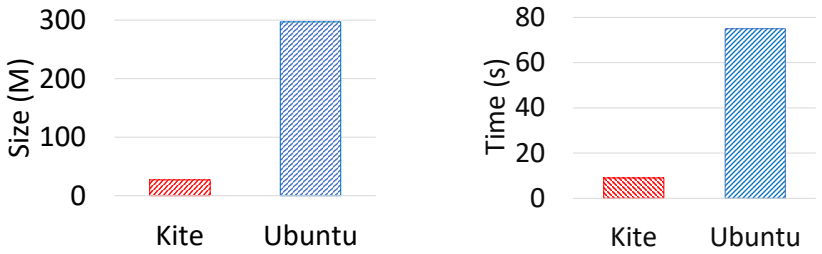
In Section 10.1, we compare Kite's executable size and boot time to those of other operating systems capable of running driver domains. Our analysis of different gadgets in Sections 10.2 and 10.3 shows that Kite has a lower gadget quality and quantity measured in various matrics, making it less vulnerable to attack compared to other state-of-the-art solutions. Finally, in Section 10.4, we assess Kite's resilience to known vulnerabilities, considering that it requires a minimal number of system calls, which could compromise driver domains.

10.1 Image Size and Boot Time

We compare the image size of Kite with Ubuntu 18.04, the Linux distribution we used in the presented experiments. We measured the size of the entire Kite network domain binary. For Linux, we measured only the size of the kernel and its modules, i.e., we did not include

the size of user-space programs and libraries such as libc. As Figure 10.1a shows, the Linux image is about 10x larger than the Kite image.

Since the boot time directly affects deployment in the cloud infrastructure, it is crucial to reduce it as much as possible. Moreover, driver domains may need to restart to recover from failures or attacks, where faster boot times are equally important. As Figure 10.1b shows, Kite only takes 13 seconds to boot the network domain. In contrast, Ubuntu needs 75 seconds.



(a) Image size. (b) Boot time.

Figure 10.1: Image size and boot time comparison.

10.2 ROP Gadget

The number of ROP gadgets is one concrete, quantitative metric that can be used to evaluate security. A smaller number of gadgets indicates potential obstacles for an attacker since the attacker will have a hard time finding appropriate gadgets to exploit a known vulnerability [46]. Moreover, the attack surface is proportionally reduced in Kite, which also helps in achieving better security. Using the methodology from [71], we count gadgets belonging to different categories: Data move, Arithmetic, Logic, Control flow, Shift & Rotate, Setting flags, String, Floating point, Misc, MMX, NOP, and RET. Each category represents a class of operations.

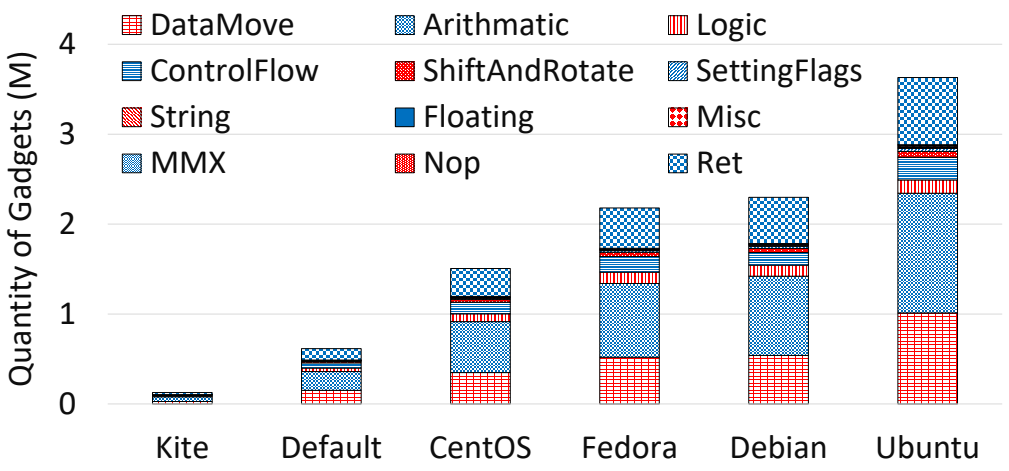


Figure 10.2: ROP gadget comparison.

Figure 10.2 shows the quantity of ROP gadgets for the Kite, vanilla Linux, and popular Linux distributions such as CentOS 8, Fedora Rawhide (05/2020), Debian 10.4, and Ubuntu 18.04. For Kite, we measured the number of instructions from different categories present in the Kite network domain image. For the vanilla Linux kernel, we did the same for the kernel image built with the default configuration (defconfig), which is a fairly minimalistic configuration. The vanilla image alone already has four times more gadgets than the Kite image, but for using Linux as a network domain, we generally need kernel modules. (Not to mention user-space libraries.) Therefore, we measured the quantity of ROP gadgets for associated kernel modules along with the kernel image for the presented Linux distributions that are capable of running as the network driver domain.

We used Ropper [24] to measure the number of ROP gadgets. We found that the number of ROP gadgets for Kite is 4x and 24x smaller than that of the Linux default configuration and Ubuntu, respectively. Kite's substantially smaller number of gadgets indicates its great potential for better security when used as a driver domain.

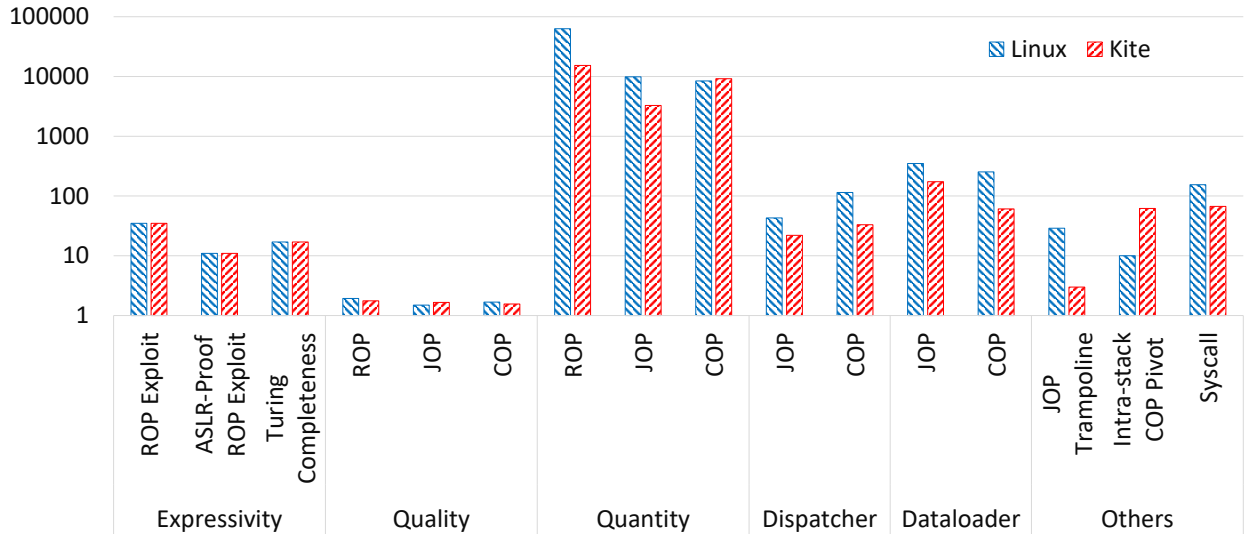


Figure 10.3: Gadget set analysis.

10.3 Gadget Set Analysis

Though Kite has significantly fewer ROP gadgets presented in its executable compared to different flavors of Linux distributions, the total gadget count may fall short of representing its resilience against attacks. Attackers may exploit remaining gadgets to manipulate an attack. Moreover, some attack surface reduction attempts can introduce newer gadgets. Brown et al. [58] proposed an in-depth gadget analysis to understand the likelihood of the program variants falling victim to gadget-based attacks compared. They offer an open-source tool named GadgetSetAnalyzer that performs static binary analysis on program variants and produces results based on the quantity, quality, and locality of code reuse gadget sets, indicating the impact of attack surface reduction. We use GadgetSetAnalyzer to compare Kite and default Linux kernel's likelihood of gadget-oriented attacks. Figure 10.3 depicts the corresponding results in logarithmic scale for different gadget utility metrics.

Abstract instructions, known as functional gadgets, are utilized to carry out fundamental computational operations like addition, register loading, and logical branching. These op-

erations are then combined to create a malicious payload. The computational capability allowed by a group of gadgets is determined by their expressivity. As shown in figure 10.3, Kite does not increase the expressivity of driver domains when fundamental computational operations are taken into account.

Kite reduces the quality of gadgets, such as gadget length and memory side effects, for ROP and call-oriented program (COP) gadgets compared to Linux. Additionally, Kite reduces the quantity of ROP and jump-oriented (JOP) gadgets. However, Kite shows a slight increase in the COP quantity and corresponding intra-stack pivot. We attribute this increase to Rumprun's introduced function calls for performing system calls, as opposed to Linux, which uses the same trampoline function for system calls because of the context switching between userspace and kernel space. Moreover, the count of gadgets that dispatch JOP and COP and the count of gadgets that load data for JOP and COP gadgets are also lower for Kite. Finally, Kite significantly reduces JOP trampoline and syscall gadgets in comparison to Linux.

10.4 Syscall Reduction and CVEs

The minimalistic design of Kite allows only a handful of libraries, selected drivers, and one application per driver domain. These applications replace the need for several userspace libraries (e.g. python) and tools (e.g. xen-tools). Therefore, Kite driver domains are safe from many known vulnerabilities, such as CVE-2016-4963 and CVE-2013-2072, associated with unneeded libraries and applications that are part of traditional service VMs. We found 172 [32] and 92 [33] reported CVEs that make use of crafted applications and shell, respectively, for performing attacks on Linux-based OSs. Being single-purpose OSs without rich user-space environments, Kite VMs prevent the attackers from running malicious applications or using shells.

Table 10.1: Examples of CVEs prevented by only keeping necessary system calls.

CVE ID	Syscall Name	Description
2021-35039	init_module	Linux kernel loading unsigned kernel modules via init_module syscall.
2019-3901	execve	A race condition allows attackers to leak sensitive data from setuid programs.
2018-18281	ftruncate, mremap	Permits access to a already freed and reused physical page.
2018-1068	compat_sys_setsockopt	Allows a privileged user to arbitrarily write to kernel memory.
2017-18344	timer_create	Allows userspace applications to read arbitrary kernel memory.
2017-17053	modify_ldt, clone	Allows an attacker to exploit use-after-free by running a crafted program.
2016-6198	rename	Allows local users to cause a denial of service attack.
2016-6197	rename, unlink	Allows local users to cause a denial of service attack.
2014-3180	compat_sys_nanosleep	Usage of uninitialized data creates possible out-of-bounds read.
2009-0028	clone	Allows unprivileged child process to send arbitrary signals to its parent.
2009-0835	chmod, stat	Allows local users to bypass intended access restrictions via crafted syscalls.

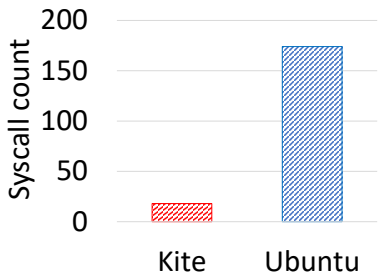


Figure 10.4: System call count comparison.

Rumprun leverages syscall-related functions from NetBSD. Since each Kite VM runs one specific application, we can easily pinpoint the system calls that are used. We found that rumprun uses 14 and 18 system calls for the network and storage domain, respectively, whereas even minimal Ubuntu-based driver domains use **10x** more systems calls (Figure 10.4). Furthermore, to prevent attackers from using other syscalls, we discard all remaining syscalls during the compilation process. This reduces the attack surface and mitigates many CVEs, including 11 CVEs presented in the Table 10.1.Though we can block a few syscalls in Linux, lots of them are essential to initialize and run driver domains and cannot be removed.

Chapter 11

Redwood: Flexible Secure VMM

VMMs play important roles in deploying and managing VMs. They ensure the isolation and security of VMs running on the same physical machine. However, there are components in a general VMM architecture that are susceptible to future vulnerabilities and may contribute to propagating and aggregating damage. Additionally, performance acceleration architectures, such as DPDK, when incorporated with VMMs, can become a single point of catastrophic vulnerability with the potential to affect multiple VMs. This chapter examines vulnerable points in VMMs and introduces the concept of Redwood, a method for specializing and compartmentalizing VMMs to make them more resilient to future attacks by mitigating these vulnerabilities. It also discusses strategies for securing acceleration architectures in the virtualization ecosystem.

Sections 11.1 and 11.2 describe Redwood's design principles and associated challenges, respectively. The design for achieving VMM specialization is discussed in Section 11.4, and the design for establishing compartmentalization is described in Section 11.5. Finally, Section 11.6 covers isolation for vhost-user interfaces to secure VMs connected to OSS-DPDK.

11.1 Design Principles

Existing VMM designs have several limitations that make the virtualization infrastructure prone to common vulnerabilities. These vulnerabilities include memory manipulation, ar-

bitrary code execution, denial of service, etc. Attackers can exploit these vulnerabilities to access sensitive information, disrupt services, and cause financial losses. To address these issues, we propose a set of new design principles for VMMs that can enhance security in cloud infrastructures. These principles focus on per-VM specialization and fine-grained intra-VMM component compartmentalization.

(P1) ***The design of a VMM should be modular and highly configurable***, allowing for the removal of any default features or implementations that are not desired. A highly flexible VMM is a prerequisite for creating a minimal VMM that can be specialized to meet the specific needs of each VM.

(P2) ***A VMM should not have any unnecessary components.*** Each VMM should be tailored to meet the specific needs of individual VMs, rather than being designed as a generic, full-featured VMM. This approach helps to create VMMs resilient against vulnerabilities by minimizing their attack surface.

(P3) ***Each pair of unrelated VMM components should be kept in separate compartments.*** This approach helps limit an attacker's freedom of using vulnerabilities in one component as a means of exploiting another. Compartmentalized components make it more difficult for attackers to traverse the whole VMM to gain access to sensitive information or disrupt operations.

(P4) ***Each instance of the same component type should be isolated from one another.*** Although there may be multiple instances of the same component type, keeping them in separate compartments helps to avoid a vulnerability in one instance propagating to the other and improve the overall security of the VMM.

(P5) ***Security should not come at the cost of performance.*** It is crucial to balance the two. Therefore, isolation mechanisms should be chosen carefully so that they do not neg-

atively impact performance. These mechanisms should be lightweight and efficient, allowing the system to maintain optimal performance while providing the necessary protection.

(P6) ***The compatibility with the existing VMs should not be compromised.*** Introduced security primitives should be transparent to the VMs, so that they can continue to function as normal while the VMM enforces security. This allows for the best of both worlds, providing a secure environment while minimizing the impact on existing systems.

11.2 Challenges

VMMs run on the host OS and are considered the TCB of the cloud infrastructure. Though there are several VMMs available with a wide range of features, they lack flexibility in terms of configuration. This means that VMs must use VMMs that are built to be generic and include many default features unrelated to the VM, resulting in a larger attack surface. This presents the challenge of designing and implementing a highly modular and configurable VMM, which is crucial for minimizing the TCB in cloud infrastructure. Building a modular and flexible VMM from scratch requires significant implementation effort and expertise. On the other hand, the rigid design of generic VMMs makes it challenging to achieve a minimal, specialized VMM for individual VMs.

When a VMM is deployed, the entire process is considered to be trusted. However, even if unwanted features are removed, the VMM's security can still be compromised due to existing bugs in any part of the remaining implementation. Current VMMs lack security primitives to enforce compartmentalization between different VMM components, making them vulnerable to attacks. Exploiting a vulnerability in one part of the VM can allow an adversary to manipulate another part. Implementing compartmentalization in a VMM requires careful design and implementation techniques. Deciding when to enable and disable

access to a compartment requires thorough examination of interactions between different parts of the VMM. To the best of our knowledge, Redwood is the first attempt to deploy VMM components in separate compartments for improving cloud security.

Isolation has been a topic of active research for years, and there are several software- and hardware-based isolation techniques, each of which have pros and cons in terms of security and performance. Some isolation techniques offer improved security at the cost of performance, while others prioritize performance over security. Additionally, not all existing isolation techniques are suitable for intra-VMM isolation. As a result, finding an isolation technique that achieves the desired level of compartmentalization within a VMM without sacrificing VM performance is a challenging problem. We explore both software and hardware solutions to balance performance and security in a virtualized environment.

Introducing flexibility in VMM configuration and compartmentalization between VMM components deserves a thorough re-evaluation of the design and a significant amount of additional effort in implementation. A poorly designed VMM may have limited capabilities, potentially rendering it inadequate in serving existing VMs because of missing features. Moreover, modifying existing VMs to be compatible with the proposed VMM is often not desirable. Thus, it is crucial to carefully consider the compatibility of new security primitives with existing VMs when selecting isolation mechanisms and specialization techniques for a VMM to ensure seamless integration.

11.3 Trusted Computing Base and Trust Model

The virtualization stack has several components that are critical to Redwood's functional correctness, including the host OS, hypervisor, compiler, linker, and loader. Any compromise of these trusted components would compromise Redwood's correctness.

The VMM is consists of various components, including core and additional components (see details in 11.4). The core components are relatively small and potentially formally verifiable. In contrast, there are many additional components and we consider all of them to be mutually distrusting, as each component may have potential vulnerabilities that an adversary could exploit to attack other parts of the VMM.

The Redwood prototype (discussed in Section 12) leverages the Intel MPK [67] hardware feature for compartmentalization. We trust the MPK APIs for enabling and disabling protection keys and assume that they work as intended.

11.4 Achieving Per-VM Specialization

This section discusses various options for creating a highly configurable VMM (**P1**) that offers a minimal attack surface (**P2**) while still maintaining needed compatibility (**P6**).

VMMs have several **core** responsibilities, including creating, allocating resources for, launching, and destroying VM. However, these tasks are not sufficient for running an operating system with workloads inside a VM. As a result, a VMM may need to provide **additional** support, such as firmware like BIOS and UEFI, interrupt controllers like APIC, MSI, and MSI-X, configuration and power interfaces like ACPI, I/O devices, and migration mechanism.

Any additional feature not needed by the workload only contributes to the attack surface. Therefore, the uKVM [138] approach combines unikernels with relevant VMM components and replaces standard PV devices with low-level interfaces to create a unified executable. However, this approach is designed specifically for and requires modifications to unikernels, which may violate the principle (**P6**) of backward compatibility and may not be suitable

for all VMs. In addition, running a user-provided executable directly on the host poses a security risk to the ecosystem.

The virtualization community often introduces new software features, such as vDPA [4] and VFIO [5], and hardware features, such as Intel SGX [69] and AMD SEV [35], to improve performance and security. Full-fledged VMMs like QEMU are often the first to test these features due to their widespread market share. However, build-from-scratch approaches like uKVM suffer from a lack of support and infrequent updates. Therefore, even if a workload desires, it may not use such minimal VMMs because of missing features.

In contrast, QEMU typically has hundreds of features implemented. Firecracker [118] and Cloud Hypervisor [45] selectively implement fewer features as they aim only to support microVM (lightweight Linux) and cloud OSs, respectively. All these VMMs make most of their features default and non-removable. However, not all workloads have the same requirements. To create a VMM with minimal attack surface, it is necessary to make each VMM highly customizable to suit the specific needs of different workloads.

Achieving a minimal VMM from an existing one losing no necessary features or compatibility requires fine-grain configurability. This requires a nontrivial modularization and refactoring of the current implementation. After identifying all the additional components, a refactoring is necessary to ensure that each fine-grained component can function independently along with the core parts. Finally, we need to create a configuration space with one configuration per component, allowing users to choose which components to include at build time. This can be achieved using platform-specific tools or techniques such as the Kconfig language, the *define* keyword in C, or the *features* feature in Rust.

We propose creating various VMM images for different configurations, rather than using the same image for all VMs or receiving different images from clients. Then, based on the

requested VM configuration, the corresponding VMM image can be used to run the VM. Our prototype suggests that this approach can reduce the size of the VMM by at least 50%. Therefore, the additional disk space required for storing the executables will not be significant. Next, we discuss features we can take into consideration to make them highly configurable at build time.

11.4.1 VM Bootloading

VM bootloading is the process for a VM to start up and loads its OS. Bootloading a VM is similar to bootloading a physical machine, but with some differences because of the virtualized environment.

The boot process begins with the firmware, which locates the Master Boot Record (MBR) on the VM image and loading the bootloader. The bootloader is responsible for loading the kernel, which is the core component of the OS. The kernel then starts the init process, which initializes and running the rest of the OS.

There are several ways to boot a VM, including:

- Boot like a regular machine, where the firmware locates the MBR on the VM image and loads the bootloader from there. Then, the bootloader loads the kernel. Finally, the kernel starts the init process.

- VMM directly loads the GRUB bootloader from the host using the configuration from the VM. The GRUB then loads the kernel, which starts the init process.

- A kernel and an init disk can also be directly supplied to the VMM, allowing it to skip many of the normal steps in the boot process. Therefore, the VMM directly loads

Table 11.1: Example of emulated and paravirtualized devices in VMMs.

Emulated device	**Paravirtualized Device**
Serial port	Virtio console
Network card (E1000, NE2000 PCI)	Virtio network, Xen netback
ATA/SATA, CD/DVD-ROM, floppy disk	Virtio block, Xen blkback
Graphics card (CL-GD5446, Red Hat QXL VGA)	Virtio GPU
PS/2 mouse and keyboard, HID	Virtio mouse and keyboard
SCSI controller (LSI53C895A, NCR53C9x)	Virtio SCSI
USB controllers (UHCI, EHCI, xHCI)	Xen USB
Watchdog timer (Intel 6300 ESB PCI, or iB700 ISA)	Virtio watchdog

the kernel and the init disk from the host. The kernel starts the init process from the supplied disk.

As there are a few other ways to boot a VM's operating system, a VMM usually supports multiple bootloading mechanisms. However, a particular VM only needs one method. Therefore, a VMM can minimize its functionality by keeping only the necessary method.

11.4.2 Virtual I/O Device

A VMM can facilitate I/O operations by a PV, a fully emulated, or a physical device. For example, a VM can use an emulated e1000 NIC, a virtio network device, or a physical NIC accessed with PCI-passthrough. Table 11.1 lists a subset of emulated and PV devices supported by QEMU.

Each device type can have further variations and a full-fledged VMM provides a wide range implementations to accommodate these variations. For example, a virtio block device may support different virtual disk formats, such as raw, qcow, vdi, and vhdx. Figure 11.1 high-

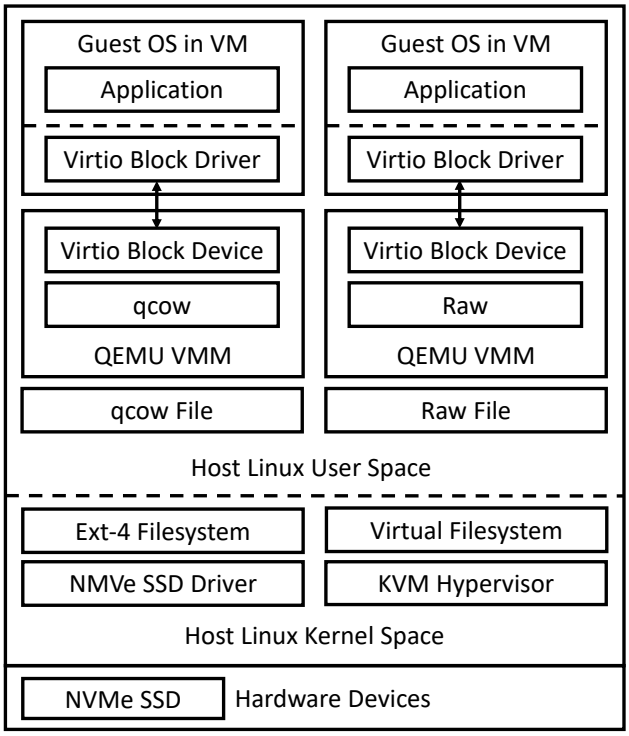


Figure 11.1: An example of a virtualization stack focusing on components for VMs using PV storage. The host OS, such as Linux, manages the physical storage device (e.g. NVMe SSD) driver, a filesystem (e.g., Ext-4), a virtual filesystem providing a common interface, and a hypervisor (e.g., KVM). The virtual disk images are files of specific formats (e.g., raw and qcow) in the host OS. VMM (e.g., QEMU) instances provide implementations for these formats and PV block devices (similar to blkback in Xen). A guest OS runs the PV block driver and filesystems in its kernel space for offering storage interfaces to applications.

lights storage components in an example of a virtualization stack for running a guest VM that uses PV storage. However, a VM typically only requires a limited set of these implementations to facilitate its workloads. For instance, a VM running a web server may only need a network, a storage, a serial and a clock device, making the use of other I/O devices present in the VMM unnecessary.

Figure 11.2 illustrates the count of I/O device-related and the total number of CVEs reported for QEMU over the years from 2007 to 2021. As the graph indicates, I/O devices account for approximately one-third of the total number of reported CVEs. This suggests that I/O device

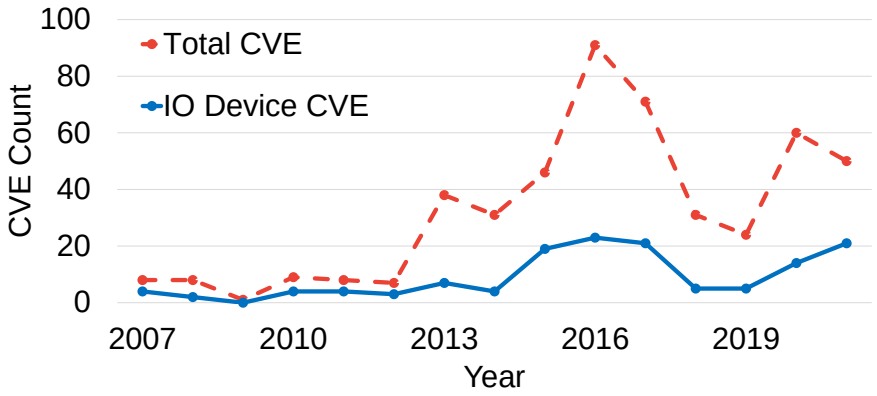


Figure 11.2: QEMU CVE trend from 2007 to 2022. [41]

implementations may have a higher number of vulnerabilities compared to other parts of a VMM. This is an important consideration when it comes to security, as these vulnerabilities can provide an attacker with a potential entry point to the system.

To mitigate this risk, one effective strategy is to reduce the number of I/O implementations by retaining only those that are needed by a particular VM. This helps to significantly reduce the VMM's attack surface by eliminating unnecessary or unused I/O device implementations.Therefore, it is essential to make the I/O device implementations highly configurable when designing a VMM so that one can discard all unrelated devices at build time.

11.4.3 VM Migration

The capability of migrating VMs between different hosts is a crucial feature for managing various operations in the cloud, such as workload balancing and system upgradation. A VMM may have multiple migration techniques implemented, such as checkpoint restart, post-copy, pre-copy, and hybrid migration. Each technique has its advantages and disadvantages, and a VM may require a specific technique that best suits its purpose when migration is necessary. Some short-lived VMs may not require migration at all during their lifespan. For example, a VM providing function as a service (FaaS) may boot quickly and shut down immediately

after serving the requested function. Therefore, it may not need VM migration. Depending on a VM's purpose, reducing or eliminating migration-related implementation from a VMM can help reduce the attack surface and improve security.

11.5 Establishing Intra-VMM Isolation

VMM instances run as userspace processes in the host OS, contributing to TCB of the ecosystem and increasing the attack surface. While reducing the size of the VMM can significantly reduce this attack surface, vulnerabilities in the remaining implementation can still pose serious security risks. For example, if a VM requires multiple I/O devices and one of them has a bug that allows an attacker to gain privilege of the VMM process, the attacker may disrupt the execution flow or steal data from this device or other parts of the VMM. Isolation can prevent an untrusted component from directly accessing the private memory of other components. Partitioning sensitive data and code into isolated components within the VMM can help limit the impact of an attack and preserve the confidentiality and integrity of other components' data.

The approach of intra-VMM isolation requires a careful examination of the VMM design to determine which components need to be isolated into separate compartments and what the isolation boundary should be. Therefore, we need to answer the question of what to isolate. Not all components can be compartmentalized in the same manner due to differences in functionality and implementation. Additionally, there may be multiple isolation techniques available, depending on the hardware and software environment, leading to the question of how to isolate. Finally, based on the chosen isolation technique and the functionality of the isolated components, decisions must be made regarding when to enable or disable access to a compartment and who has the authority to do so. In the following sections, we present

our analysis to address these questions.

11.5.1 What to Compartmentalize?

To compartmentalize a VMM, the first step is to identify which components in it we want to isolate. For instance, virtual I/O devices are a major part of any VMM and often share vulnerabilities, as shown in Figure 11.2. Therefore, virtual devices are potential candidate for compartmentalization. According to the principle **P4**, there should be isolation between multiple instances of the same type of device in a VMM. For example, if a VM requires two network devices, the VMM should isolate both instances to ensures that a vulnerability in one network devices cannot propagate to the other device.

11.5.2 How to Compartmentalize?

Prior protection key-based isolation approaches [77, 90, 124, 131, 134] draw isolation boundaries around libraries and the application code. For instance, there can be multiple compartments in FlexOS [90], where each can contain one or more libraries. When a compartment needs to execute a function in another compartment, the caller compartment's memory is disabled, and the callee compartment's memory is enabled. Upon return, memory permission is changed in reverse order. This model does not exactly fit for isolating VMM devices, which are not necessarily libraries, and they do not get invoked like library functions. Unlike functions performing operation on the supplied arguments, each device instance has its own memory with sensitive information, which needs to be secured. Moreover, I/O devices perform asynchronous operations. Therefore, keeping other I/O devices disabled when one is performing is not a viable option. This scenario demands fine-grained isolation. For example, during a device instantiation, a VMM may create a unique protection key to label every

memory location that belongs to and represents this instance. Any access to these memory locations will require proper permission change, otherwise will be considered illegal.

Technologies such as Intel MPK [67] and ARM MTE [1] can be utilized to implement such isolation model. Hardware capabilities, such as the CHERI [136] ISA extension for ARMv8-A, can also be used to assign different capabilities to different components to compartmentalize the VMM. However, TrustZone [94], AMD SEV [35], and EPT/VM all utilize compartments as separate systems, communicating through RPCs and shared memory. Therefore, using them to isolate VMM components may result in significant performance penalties, which goes against the principle **P5**. Additionally, Intel SGX is not capable of performing I/O, so isolating device drivers in this environment is not feasible.

11.5.3 When to Enable/Disable a Compartment?

Our goal is to keep the memory in a compartment disabled when it is not being used. Therefore, a vulnerability in another part of the VMM cannot propagate and arbitrarily execute device operations. Since I/O device operations are asynchronous events, a dedicated (not shared) event handler can enable and disable memory accesses if protection key-based techniques are used.

11.6 Securing OVS-DPDK Vhost User

The vhost-user model allows offloading virtio device implementation from the VMM to a userspace application running on the host. Moving the backend in a separate process is supposed to increase security by providing process-level isolation, incurring a possible performance penalty because of added indirection. Upon integrating the vhost-user with the performance acceleration platforms, like DPDK, the performance increases significantly.

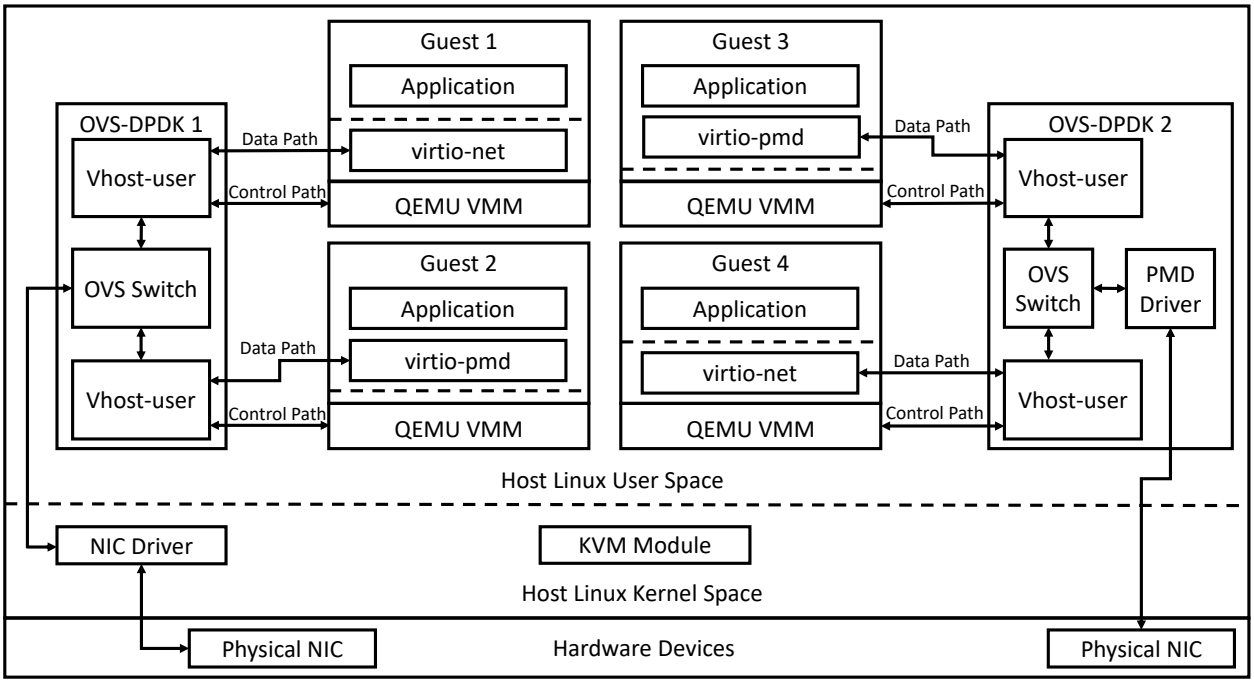


Figure 11.3: The OVS-DPDK architecture provides faster networking for guest VMs by running as a userspace process on the host and utilizing vhost-user interfaces and an OVS switch. It connects to physical NICs through the NIC driver in the kernel or the PMD driver in DPDK and establishes the control path with the VMM and data paths with the guest VMs through virtio-net or virtio-pmd drivers. However, it also poses a security risk as the OVS-DPDK process has unfiltered access to all vhost-user instances, which may contain sensitive information from the guest VMs.

However, it creates an opportunity for a threat of higher magnitude than the security gain from process-level isolation. Here, we discuss the scope of vulnerability in DPDK and the design for mitigation.

Figure 11.3 depicts how VMMs can connect to the OVS-DPDK process through vhost-user interfaces. OVS-DPDK creates vhost-user instances for each guest VM connected to this application. To establish direct memory access between the vhost-user backend and the virtio frontend, the VMM shares the whole VM's memory with the OVS-DPDK. That means the OVS-DPDK has direct access to the memory of all VMs connected to it. It makes the OVS-DPDK a lucrative hub for intruders to manipulate attacks and disrupt all VMs' execution or steal their data.

There are multiple scenarios of how an intruder can gain access to OVS-DPDK. Any vulnerability on the host OS, VMM, OVS, or DPDK itself that gives an attacker full or partial access to DPDK provides open access to the VMs' memory connected through the vhost-user interface. For instance, CVE-2020-14377 [14] shows that a buffer over-read vulnerability in DPDK can let a VM user do unauthorized access to the host's memory. CVE-2020-14377 [14] shows that an attacker can exercise a buffer overflow attack to write arbitrary data to any address in DPDK. Similar attacks may lead one VM user to read other VM memory since DPDK already has access to all connected VMs' memory.

We propose to scrutinize OVS-DPDK's access to the guest VM's memory. One idea is putting each VM's memory in a separate compartment and enabling access to that compartment just for the corresponding vhost-user. Any activity of undesired access to a compartment should be identified as an attack. For example, accessing a memory address using buffer overflow or arbitrary code execution should trigger an action, like a segmentation fault or application reset, that would prevent the attacker from proceeding further. Isolation techniques such as memory tagging, capability, etc., can be used based on the security and performance metrics.

Chapter 12

Implementation of Redwood

The design principles and techniques of Redwood, outlined in Chapter 11, can be applied to all existing general-purpose VMMs. We developed a prototype for Redwood based on Cloud Hypervisor, an open-source VMM that runs on KVM and Hyper-V hypervisors. This chapter covers the implementation details of Redwood toward creating a minimal VMM from an existing VMM and isolate PV I/O devices using hardware-enforced techniques. To enhance the virtualization ecosystem, Redwood has added unikernel bootload support and compartmentalized memory access for VMs using OVS-DPDK.

Section 12.1 explains how Cloud Hypervisor was transformed into a highly flexible VMM, enabling customization to meet specific requirements. Section 12.2 delves into the technical details of compartmentalizing I/O devices using hardware support from Intel's MPK. Section 12.3 outlines the multiboot implementation for Redwood. Finally, Section 12.4 explains how memory compartmentalization for each VM, connected to OVS-DPDK via vhost-user-net interfaces, is implemented to restrict attacker access.

12.1 Workload-aware Redwood

Redwood aims for improving VMM better security in cloud infrastructure. Cloud Hypervisor is suitable for our target problem space because it is designed to run modern cloud workloads. It can run cloud OSs with most I/O handled by virtio devices and 64-bit CPUs. We chose

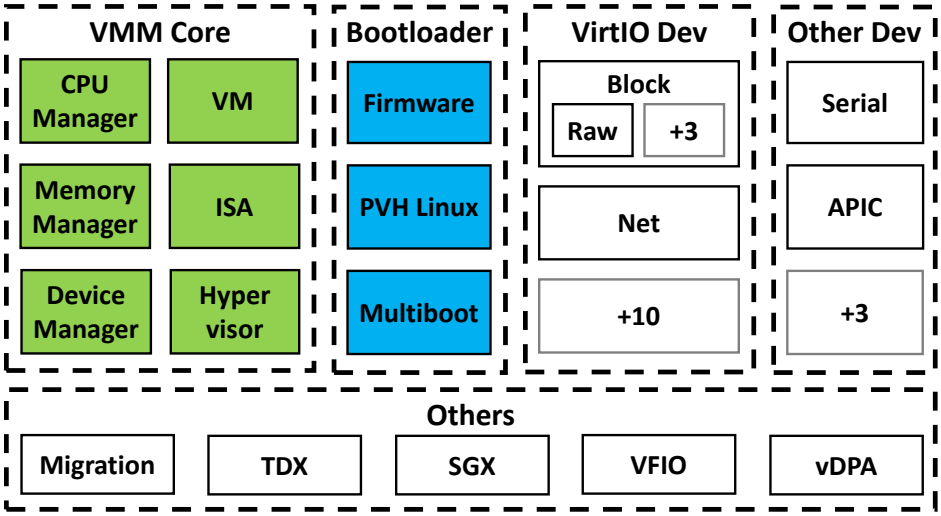


Figure 12.1: Core and additional components of Redwood.

Cloud Hypervisor as a base for Redwood's prototype over QEMU because it is written in Rust, which is programming language known for providing better memory safety than C. Although Firecracker is also written in Rust, we chose Cloud Hypervisor for its broader support for general cloud OSs, rather than a specific configurations like micro-VMs.

Rust's package manager, Cargo, provides a mechanism called *features* for conditional compilation and optional dependencies. As discussed in Section 11.4, we identify the core and additional components in Cloud Hypervisor and refactor all additional components into named *features* in Redwood. Therefore, users can enable and disable components at build time by choosing which to keep in Redwood image.

Figure 12.1 shows Redwood components. Green boxes represent core components, which include CPU, memory, device managers, and APIs for VM, ISA, and hypervisor operations necessary for running any VM. These managers create and manage VCPUs, memory, and devices for VM. VM APIs perform VM creation, launching, destruction, etc. ISA (x86-64 and AArch64) and hypervisor (KVM and Hyper-V) supports were already configurable in Cloud Hypervisor.

Cloud Hypervisor supports booting OSs using third-party firmware, such as OVMF [2] and Rust Hypervisor Firmware [3], and direct booting uncompressed Linux into 64-bit mode. Along with these two, Redwood adds multiboot support (Section 12.3). As discussed in Section 11.4.1, a VM needs only one booting mechanism. Separating these three bootloaders into features (blue boxes), Redwood provides the flexibility to include only the required one at build time.

Cloud Hypervisor implements 18 I/O devices. Along with the virtio devices, listed in Table 12.1, it implements ACPI, IO APIC, and legacy devices such as RTC/CMOS, GPIO, and serial. Only RTC/CMOS is configurable at build time. Redwood resolves any interdependency, if any, between all these I/O devices and makes them configurable at build-time. We also make disk implementations configurable, allowing a VMM to have specific types needed for a particular VM. Supported formats include raw, qcow, VHD, and VHDX.

Cloud hypervisor supports security features, like Intel SGX [69] and TDX [16], and device passthrough mechanisms like VFIO and vDPA, where TDX and VFIO are configurable. It also supports live migration as a default feature. Redwood separates the VM SGX, vDPA, and migration-related sources and makes them optional feature. All additional components in Redwood are marked with white boxes.

12.2 Isolation inside Redwood

Cloud Hypervisor implements 13 virtio devices, listed in Table 12.1. Redwood isolates all instances of these devices, i.e., if there are multiple instances of the same device (say, two block devices), each will be isolated into separate compartments.

Cloud Hypervisor maintains per-device information, which includes a list of features, various

Table 12.1: Isolated virtio PV devices in Redwood.

Device Name	Description
virtio-net	Ethernet card for networking
virtio-block	Block device for storage volume
virtio-console	Console device for data input-output
virtio-balloon	Device used for memory ballooing
virtio-mem	Device for hot (un)plugging memory
virtio-pmem	Device for resizing page cache
virtio-rng	Device for random number generation
virtio-watchdog	Device for monitoring system status
virtio-iommu	Device for DMA management
virtio-vsock	Device for guest-host communication
vhost-user-net	Network device in separate process
vhost-user-block	Storage device in separate process
vhost-user-fs	Device for sharing host directory

events and interrupt handlers, and the driver (in VM) detail. Each type of device has additional device-specific information. For instance, a block device has unique ID, information of underlying disk, block device configuration, which are all sensitive and crucial to perform a virtio-block device operation. The VMM populates this information during the instantiation process of a device.

On the other hand, the device and drivers perform asynchronous I/O operations. Therefore, Cloud Hypervisor creates a distinct event handler thread per device instance to handle corresponding I/O events. The event handler has access to much sensitive information regarding the virtio queue, VM's memory, interrupts, events, and the underlying device interface and configuration. With access to all mentioned information, an intruder has a lot of flexibility to manipulate an attack to disrupt execution, or steal data. To limit the

exposure of sensitive information, Redwood labels each memory location that belongs to the device with a unique memory protection key during the instantiation process.

Redwood utilizes Intel's MPK [67] feature to label memory pages and control access rights with protection keys (pkeys). For every access to a tagged memory page, an additional permission check is performed using the PKRU register, which has access disable and write disable bits. System calls such as pkey_alloc and pkey_free are used to allocate and free pkeys, respectively. The pkey_mprotect system call is used to label a page with a key. In contrast, userspace functions for getting and setting permissions (pkey_get and pkey_set, respectively) have a low overhead, as they do not require context switching and execute in a few cycles.

As the last step of the instantiation process, Cloud Hypervisor creates an event handling thread, which waits for the I/O events from the associated driver in the VM. In such event, the handler performs corresponding I/O operations and goes to the wait state after completion. Before each time the handler goes to the wait state, Redwood disables access to all tagged memory locations by changing the permission and enable it only when an I/O event occurs. It helps reduce attacker's freedom on performing an attack using the device's memory.

Any part of the VMM can modify the value of the PKRU, thus we have to prevent unauthorized writes. Similar issue is previously solved via runtime checks [77] and static analysis [131]. Since Rust builds everything statically, except for glibc, no code is loaded after compilation in Redwood. Therefore, static binary analysis coupled with strict W⊕X is sufficient. If a VM needs more that 16 device instances, which is unlikely for a cloud workload, we can leverage the technique for pkey virtualization, devised by LibMPK [107].

12.3 Unikernel Support

The emergence of unikernel is popularizing the concept of minimalist single-purpose VMs to improve the security of cloud applications. Benefiting unikernels with the latest available features in Cloud Hypervisor and our minimalistic and compartmentalized model will contribute to improve cloud security. However, most unikernels only support multiboot for bootloading. On the other hand, Cloud Hypervisor focuses on running regular cloud OSs and does not have multiboot support. Therefore, we extend Redwood's support for unikernels by implementing a multiboot bootloader. As a result, unikernels can now benefit from our specialized hardened VMMs and, altogether, can contribute to reducing the overall virtualization attack surface.

Redwood first checks if the provided OS image supports the multiboot protocol reading the OS image. Then, it loads the kernel's text, data, and bss sections into the guest VM's memory from the image. Next, it writes some multiboot information (MBI) in VM's memory so that the OS can retrieve this MBI during execution. The bootloader also sets the GDT and IDT table with the segment registers according to the multiboot specification. Finally, it configures other VCPU registers with multiboot magic, kernel entry point, MBI's start address, and flags. Redwood avoid any BIOS interaction to make the mutiboot boot process simpler and faster.

12.4 Isolating Vhost User in OVS-DPDK

We can compare the vhost-user communication to the client-server model, where VMs are the clients and the OVS-DPDK is the server that contains multiple clients' information. The OVS-DPDK stores all the information of the corresponding drivers and the memory of VMs.

Putting the DPDK library or some part of it in a separate compartment, like inter-domain isolation approaches [77, 90, 124, 131, 132] suggest, will not work because this compartment will still have the same access to all VMs' memory. Therefore, we isolate each VMs memory inside the vhost-user driver.

We tag each VM's memory with distinct pkeys and keep the access permission on all VM's memory disabled by default. Access to the corresponding VM's memory is enabled only when a vhost-user needs to operate for a specific client and is disabled immediately once the operation is completed. As a result, the attacker cannot perform an arbitrary read/write on a guest's memory even if gets access to OVS-DPDK. We implement the permission change operations as inline functions to avoid the overhead of a runtime abstraction interface. We can use a control-flow integrity check to ensure the legitimacy of permission change operations on the tagged memory.

12.5 Implementation Effort

Converting the Cloud Hypervisor into a highly flexible VMM and compartmentalizing 13 virtio devices with Intel MPK for prototyping Redwood involves most code changes and addition. This prototype also extends Redwood's support for multiboot bootloading. Finally, we compartmentalize VMs' memory information in OVS-DPDK with Intel MPK. Table 12.2 summarizes the implementation effort for prototyping Redwood.

Table 12.2: LOC changed or added for Redwood prototype.

Component	**Description**	**LOC**
Modularization	Making highly configurable VMM	1393
Compartmentalization	Isolating virtio device and instances	1642
Multiboot	Adding multiboot support for unikernels	586
OVS-DPDK	Isolating VMs' memory at vhost-user-net	825
Total		**4446**

Chapter 13

Redwood's Performance Evaluation

The design of Redwood emphasizes security enhancement without sacrificing performance. In this chapter, we assess the performance of the Redwood prototype, which is developed based on Cloud Hypervisor (v23.0). The performance evaluation evaluates the impact of our changes on execution overhead. For comparison, we ran the original Cloud Hypervisor VMM with Linux VMs on the KVM hypervisor. Our results indicate that Redwood is equally performant as the original Cloud Hypervisor in most scenarios.

Section 13.1 outlines our setup for evaluating Redwood's prototype. Sections 13.2 and 13.3 present Virtio network and storage device performance evaluations, respectively. Redwood's virtio balloon device performance evaluation is discussed in Section 13.4. Finally, OVS-DPDK network performance evaluation for compartmentalized vhost-user-net is presented in Section 13.5.

13.1 Experimental Setup for Performance Evaluation

Table 13.1 shows the hardware configurations. In our setup, a client machine is connected directly by an SFI/SFP+ network cable to a server. The server runs the KVM hypervisor that hosts guest VMs, while the client generates load for network-related tests. Both the client and server machines run Ubuntu 18.04.3 LTS with kernel 5.0.0-23-generic. The guest VM is assigned 4GB of memory and runs Ubuntu 20.04.4 LTS, Linux cloud 5.4.0-126-generic.

Table 13.1: Hardware configuration.

	Server	**Client**
CPU	Intel Gold 5118 2.30GHz	Core i5-6600K 3.50GHz
Cores	24 (w/ HyperThreading)	4
L1/L2	32/1024 KB per core	32/256 KB per core
L3	16896 KB	6114 KB
Memory	48 GB	16 GB
Network	Intel 82599ES 10-Gigabit	Intel 82599ES 10-Gigabit
Storage	Samsung 970 EVO Plus 500GB NVMe	N/A

We ran microbenchmarks such as iPerf3 [17], ping, Netperf [21], dd [15], and pmbw [52] for performance evaluation of PV I/O devices. We also ran microbenchmarks such as ApacheBench [127]), Memtier [22], Redis [23], and SysBench [26]. Each benchmark was run for a considerable amount of time and in several iterations, and we present the average for comparing performance between Cloud Hypervisor and Redwood.

13.2 Virtio Network Device Performance

In this section, we evaluate virtio-net in Redwood and Cloud Hypervisor using a 10GbE NIC connection between the server and client machine. We ran both microbenchmarks (iPerf3 [17], ping, Netperf [21]) and microbenchmarks (ApacheBench [127], and memtier [22]) to measure network throughput and latency of real-life applications relevant to cloud users.

13.2.1 iPerf

We use the iPerf3 [17] microbenchmark (v3.7) to measure network throughput over TCP. The iPerf3 server is run on the VM, and the client is run on the client machines. We ran

each experiment for 5 minutes and repeat them 3 times.

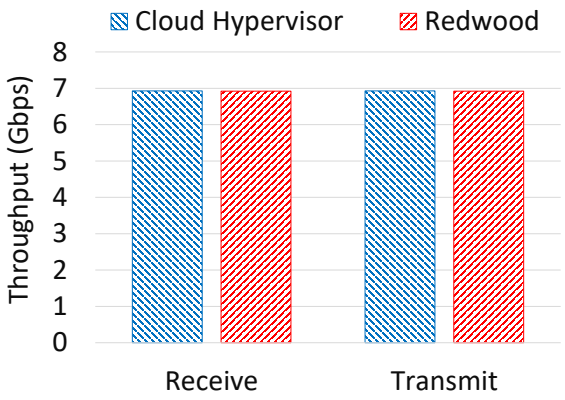
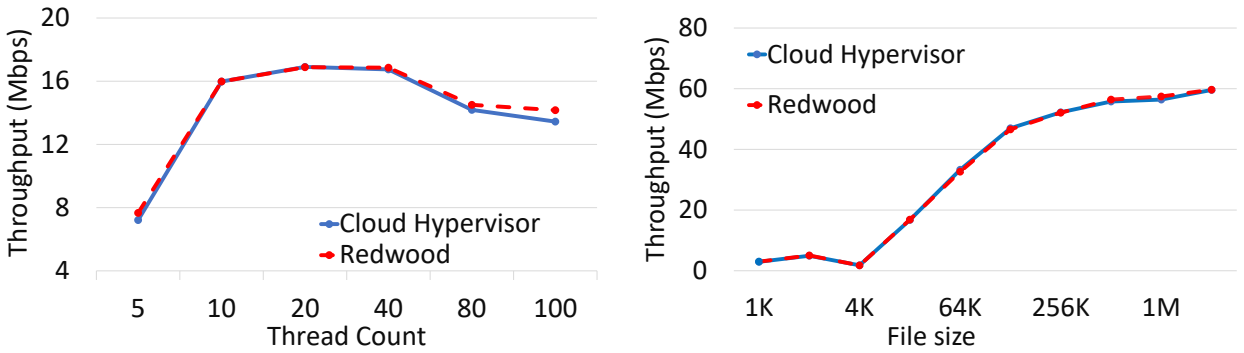


Figure 13.1: iPerf3 TCP throughputs for receive and transmit over network.

Figure 13.1 displays the receive and transmit throughputs for a VM running on Cloud Hypervisor and Redwood. As observed, the virtio-net performance is similar in both VMMs. It means MPK-based compartmentalization for PV network devices incurs no significant overhead on throughput. The RSD for these experiments are 0.0329% and 0.034% for Cloud Hypervisor and Redwood, respectively.

13.2.2 Apache

To evaluate an HTTP server, we ran Apache (v2.4.41) and the Apache benchmark (v2.3) on a guest VM and client machine, respectively. The server data consists of randomly generated files of 1KB to 1MB sizes. The benchmark sends 100,000 requests and measures the server-side throughput. Each experiment is repeated 3 times.



(a) Varying concurrency.

(b) Varying file size.

Figure 13.2: Apache server throughput measured using Apache benchmark.

Figure 13.2a shows throughput for the Apache server for various numbers of threads, ranging from 5 to 100, at the client machine sending requests for files of size 4KB. Figure 13.2b shows throughput when the benchmark requests for files of various sizes ranging from 1KB to 1MB using 40 concurrent threads. As we can see from these graphs, the virtio-net device in Redwood performs similarly to the one in Cloud Hypervisor, meaning that the HTTP server and clients can enjoy added security without losing network throughput. For these experiments, the maximum incurd RSD is 1.38% and 1.51% for Cloud Hypervisor and Redwood, respectively.

13.2.3 Network Latency

We evaluate network latency using three tools. We measure the average latency of the guest VM from the client machine using 100 pings with a one-second interval for both Cloud Hypervisor and Redwood. Netperf [21] sends 1,000 requests per second to the guest VM with even intervals. Memtier [22] benchmarks Memcached [18] in the VM by performing 100,000 SET and GET operations with a 1:10 ratio and 8KB data size.

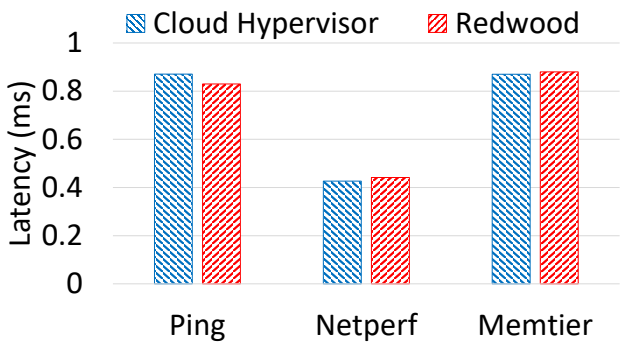


Figure 13.3: Network Latency measured using different tools.

The latency of compartmentalized and regular virtio-net devices is compared in Figure 13.3. The results show that the latency of virtio-net devices in Cloud Hypervisor (0.87ms) and Redwood (0.83ms) are similar when measured by ping. Netperf recorded latencies of 0.42ms for Cloud Hypervisor and 0.44ms for Redwood. The Memtier results showed 0.87ms latency for regular devices and 0.88ms for isolated devices. These similar results for both VMMs indicate that network device compartmentalization has no impact on latency.

13.3 Virtio Block Device Performance

For the virtio block device performance evaluation, we use an Ext-4 filesystem on an NVMe SSD attached to the host machine. A 100GB raw file on the SSD is used as a disk for the guest VM, where we ran the storage benchmarks to observe the difference between regular and compartmentalized virtio block devices. We use dd [15] as a microbenchmark and SysBench [26] (v1.1.0), a well-known file I/O benchmarking tool, as a microbenchmark. We flush the read buffer for each run and use the total I/O size (16GB) larger than the main memory (4GB) to exercise the disk aggressively.

13.3.1 dd

We use dd to perform read and write operations on the virtio block devices to measure I/O throughputs. To minimize overheads, we use `/dev/zero` and `/dev/null` devices as the read source and write destination, respectively. We transfer 1GB of data from/to the device and repeat each experiment 3 times. Figure 13.4 shows throughputs for read and write operations using virtio block devices in Cloud Hypervisor and Redwood. As it depicts, both regular and compartmentalized block devices yield similar throughput. Therefore, we can say MPK-based isolation does not negatively affect the block device throughput. The maximum RSD for these experiments are 0.012% and 0.08% for Cloud Hypervisor and Redwood, respectively.

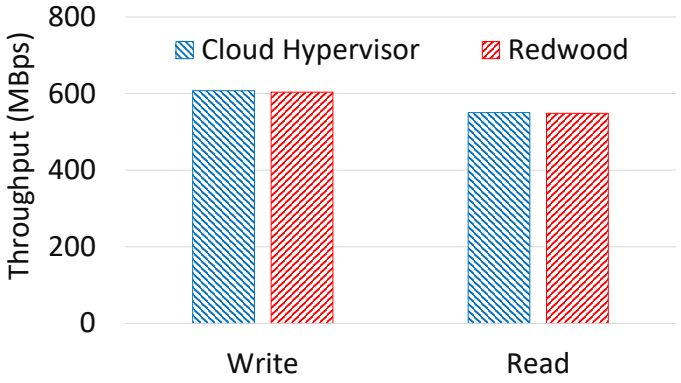
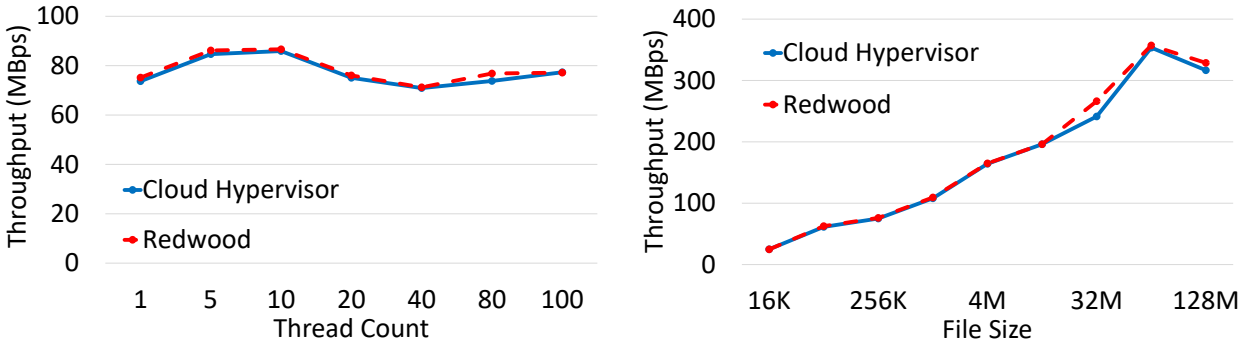


Figure 13.4: Virtio block's read and write throughput measured using dd.

13.3.2 SysBench File I/O

To measure file I/O performance, we use SysBench. It performs random reads and writes on a set of 100 files totaling 16GB. The ratio of read to write operations is 3:2, which reflects the prevalence of read operations in many real-world scenarios. We ran the same experiment with various numbers of threads (from 1 to 100) and block sizes (from 16KB to 128MB). Each run took 5 minutes.



(a) Varying number of threads. (b) SysBench file I/O throughput.

Figure 13.5: File I/O throughput measured using SysBench.

Figures 13.5a and 13.5b show throughput for a block size of 256KB varying concurrency and different number of block sizes with 20 threads, respectively. Cloud Hypervisor and Redwood has similar throughput, with average RSD of 0.73% and 0.88% and latency of 23.57ms and 25.33ms, respectively.

13.4 Virtio Balloon Performance

To evaluate the performance of the virtio balloon device on Cloud Hypervisor and Redwood, we measure memory bandwidth. We launch a VM with 4 CPUs and 10GB of memory, of which 9GB is balloon memory (i.e., not online at the beginning). We modify the Parallel Memory Bandwidth Benchmark (pmbw) [52] to vary its memory usage in each run. We gradually increase the amount of memory usage of pmbw until it reaches a limit of 8GB, where 7GB of it corresponds to balloon memory utilization. We took an average of 3 runs for each configuration.

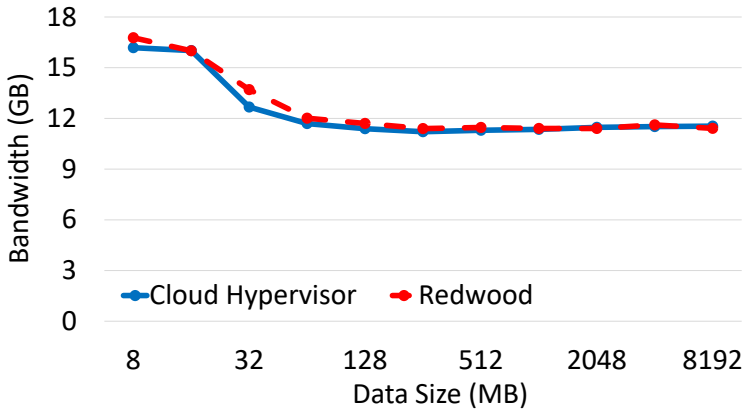


Figure 13.6: Memory bandwidth while using ballooning.

Figure 13.6 shows the memory bandwidth reported against different array sizes while using virtio balloon devices in Cloud Hypervisor and Redwood. As one can see, both regular and compartmentalized balloon devices exhibit similar memory bandwidth. The RSD for the Cloud Hypervisor and Redwood measurements are 1.77% and 1.34%.

13.5 Vhost User Network Device Performance

DPDK is known for providing better performance for networking between VMs running on the same host. Therefore, we measure network throughput, using iPerf3 [17] and Redis [23] benchmarks, of such two VMs while they are connected through vhost-user-net devices offered by OVS-DPDK.

13.5.1 iPerf

We ran the iPerf3 to measure TCP throughput for receive and transmit operations while VMs use OVS-DPDK's vhost-user-net interfaces. The server is run in one VM and the client in another. Figure 13.7 shows the network throughput for regular and isolated OVS-DPDK. As one can see, throughputs are not affected by isolation enforced on the mapped VM memory at the OVS-DPDK vhost-user side. The RSD for these experiments are 0.023% and 0.017% for regular and compartmentalized OVS-DPDK versions, respectively.

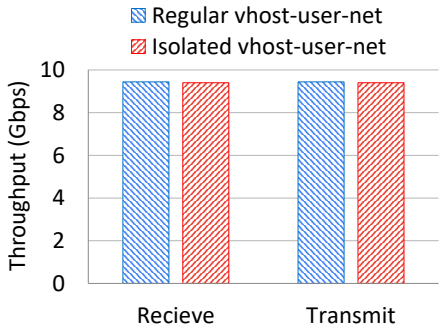


Figure 13.7: TCP receive and transmit throughput for vhost-user-net devices.

13.5.2 Redis

We utilized one VM to run Redis(v5.0.1) server, a popular key-value store. The Redis benchmark in another VM runs millions of SET and GET operations, with a pipeline size of 1000 with varying levels of concurrency, ranging from 5 to 20. Each operation performs a read/write of 128MB of data using 64-bit keys.

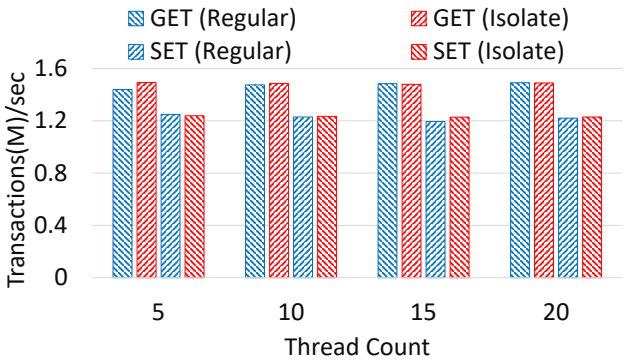


Figure 13.8: Redis throughput for operations under varying degrees of request concurrency.

Figure 13.8 shows the number of SET/GET operations performed per second while using the regular and compartmentalized version of OVS-DPDK. Overall, the regular and compartmentalized DPDK vhost-user-net exhibit similar performance. The RSD for regular and isolated DPDK is 0.00048% and 0.00051%, respectively.

Chapter 14

Redwood's Security Evaluation

In this chapter, we explore Redwood's security evaluation, which involves conducting an analysis of CVEs and assessing the VMM's attack surface. The primary objective is to evaluate how Redwood's design principles enhance the VMM's resilience against known security vulnerabilities, while also reducing the attacker's flexibility in exploiting the gadgets and bugs present in the VMM image.

Section 14.1 discusses the security aspect of Redwood in terms of CVE analysis. Section 14.2 compares the Cloud Hypervisor's VMM image size with images from different configurations of Redwood. Sections 14.3 and 14.4 present various gadget-oriented analyses with different metrics to evaluate Redwood's attack surface in comparison to Cloud Hypervisor.

14.1 CVE Analysis

Common mistakes, such as lack of validation, can create vulnerabilities such as a buffer overflow leading to severe attacks such as arbitrary code execution. A small time window between discovering a bug and applying mitigation may allow attackers to steal sensitive data or disrupt a system. However, isolation techniques can limit attackers' flexibility and prevent them from exploiting unresolved bugs to carry out attacks on a VMM. While Redwood is based on Cloud Hypervisor, the proposed hardening techniques apply to any VMM, including QEMU, and can protect against many reported CVEs. In this section, we will explore a few

of these CVEs to understand the resilience of the proposed VMM designs against them.

The VENOM vulnerability [7] (CVE-2015-3456) was discovered in 2015 but introduced in 2004. With this vulnerability, a guest VM user could keep filling a buffer in the emulated floppy disk controller in QEMU for up to 20ms, resulting in a buffer overflow. It allowed adversaries to bypass, crash, or gain code execution on the host OS with the QEMU process's privilege. Since most VMMs run with root access to the host machine, the potential damage is severe, such as affecting the execution of other VMs and allowing for the theft of their data. With our isolation approach, a VMM will deny (a segmentation fault for Redwood) any attempt to overflow into other compartments to steal or manipulate their data, as access to memory in those compartments is disabled. This way, even if a fix is not available, the severity of the attack can be mitigated or significantly reduced.

Even if a guest does not have a floppy disk configured and attached, the VENOM vulnerability is still exploitable. This is because the problem exists in the FDC, which is initialized for every guest regardless of the configuration and cannot be removed or disabled. User-level access to a guest with sufficient permissions to talk to FDC I/O ports is all that is required to exploit this flaw. Redwood creates a specialized VMM that discards features like the floppy disk if not needed for the VM. This flexibility can prevent adversaries from exploiting bugs, such as VENOM, present in default devices that are irrelevant to the VM.

We identified 128 QEMU CVEs related to different devices, including 43 related to virtio devices. For example, CVE-2021-3546 causes an out-of-bounds error when processing a specific command from a guest on the virtio vhost-user GPU device. This error could allow a privileged guest user to crash the QEMU process on the host, resulting in a denial of service or potentially enabling code execution with the privileges of the QEMU process. Enforced isolation can help prevent attackers from exploiting out-of-bounds vulnerabilities to perform arbitrary code execution using the memory of a virtio device in another compartment. We

found 79 out-of-bounds errors in QEMU. Whether QEMU is written in C, software written in memory-safe languages can also be susceptible to similar issues. For example, we found 370 CVEs related to out-of-bounds errors in software written in Rust.

CVE-2019-18960 [12] allows an attacker to carry out a buffer overflow attack on Firecracker, a rust-based VMM. The communication between the virtio device on the VMM and the driver on the VM involve message passing and a shared memory buffer. A virtio-vsock message may comprise a header, a buffer base address, and a buffer size. However, Firecracker's virtio-vsock implementation only verifies the validity of the `base` address. It does not check whether the address with `base+size` is valid or whether the address with `base` and `base+size` are within the same memory region. These flaws enable malicious guests to overflow the shared memory. Therefore, a buffer overflow on the vsock buffer can eventually affect Firecracker's heap and let an attacker steal or manipulate its memory. It is further possible for an attacker to use a heap overflow to overwrite arbitrary areas of memory, inject malicious code, and execute arbitrary code with the privilege of Firecracker process.

This CVE demonstrates that it is possible to carry out severe attacks, such as buffer overflow and arbitrary code execution, even when a VMM is written in a memory-safe language. Redwood can defend against such memory-related attacks by isolating necessary I/O devices in separate compartments and discarding unnecessary implementations. Therefore, even if the security of one device (vsock in this example) is compromised, the attacker cannot overflow to the memory that belongs to other devices and carry out an attack.

It is worth mentioning that these particular issues stem from the improper handling of raw pointers in Rust's unsafe blocks, which do not guarantee memory safety. However, often it is not possible to avoid using unsafe code in Rust. For example, Cloud Hypervisor and its dependencies contain approximately 5K lines of unsafe code.

There are a several CVEs reported for DPDK and OVS as well which are considered fatal. CVE-2020-14374 allows an attacker within a VM to exploit a buffer overflow vulnerability in the vhost-crypto DPDK application on the host. It enables the attacker to write arbitrary data to process' address and gain code execution. Though the vulnerability arises from a simple flawed bound check, an attacker can copy up to 4GB application data with full control over the copied content. CVE-2020-14375, CVE-2020-14376, and CVE-2020-14377 pose similar threats through time-of-check time-of-use, buffer overflow, and buffer over read vulnerabilities, respectively. OVS-DPDK contains guest memory addresses and mappings for all VMs connected through vhost-user. Therefore, an adversary within any of these VMs can potentially steal data from other VMs using these vulnerabilities. However, our design only allows guest memory access when needed, so an attacker will fail if they try to arbitrarily read or write to other guest's memory and will receive a segmentation fault. As a result, these vulnerabilities will likely only cause the DPDK application to crash, rather than allowing data stealing. While our design cannot completely prevent future vulnerabilities, it significantly reduces their severity.

14.2 Image Size Reduction

We compare the default Cloud Hypervisor to Redwood's various configurations based on their VMM image sizes. A basic Redwood configuration includes core VMM components, one bootloading mechanism, and no virtio devices. Adding virtio devices such as virtio-block, virtio-net, or both slightly increases the image size. As shown in Figure 14.1, the basic Redwood configuration has 50% smaller VMM image size than that of Cloud Hypervisor, and the other three Redwood configurations are also significantly more compact.

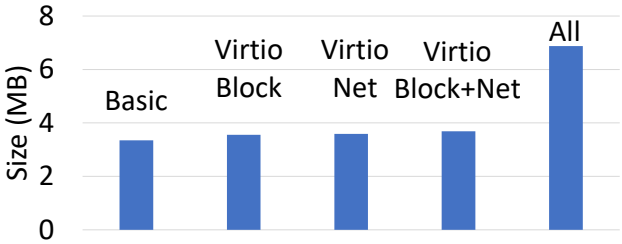


Figure 14.1: Image sizes for Cloud Hypervisor (All) and different configurations of Redwood.

14.3 ROP Gadget Reduction

Reducing the number of ROP gadgets does not always mitigate a known vulnerability. Using the methodology improve security, but a lower number of ROP gadgets can make it more difficult for an attacker to construct ROP chains to exploit a known vulnerability. Using the methodology from Follner et al. [71] and a tool [25], we counted ROP gadgets in the following categories: Data move, Arithmetic, Logic, Control flow, Shift & Rotate, Setting flags, String, Floating point, Misc, MMX, NOP, and RET. Each category represents a class of operations.

Figure 14.2 shows the breakdown of ROP gadgets by category. The basic Redwood has 3x fewer gadgets than the original Cloud Hypervisor. Redwood with common devices (virtio-net and virtio-block) also has notably fewer gadgets. This shows a great potential for improving security via VMM specialization.

14.4 Gadget Set Analysis

We perform a gadget set analysis on the default Cloud Hypervisor and different variations of Redwood configurations using the GadgetSetAnalyzer tool, like we used it for analyzing

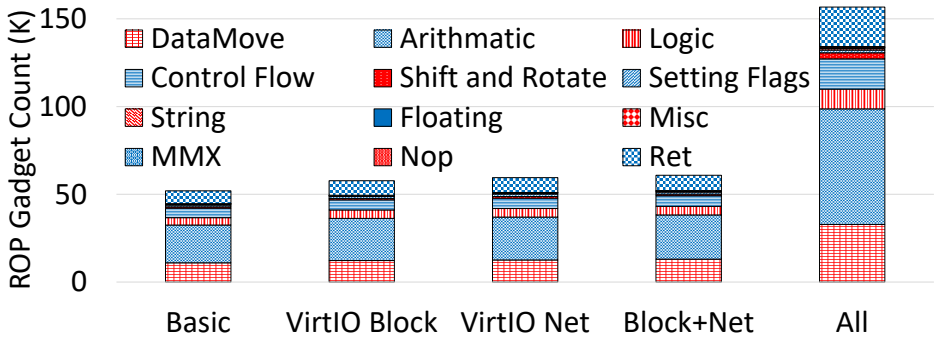


Figure 14.2: Gadgets for Cloud Hypervisor (All) and different configurations of Redwood.

Kite's gadgets in Section 10.3. The results are depicted in Fiugre 14.3.

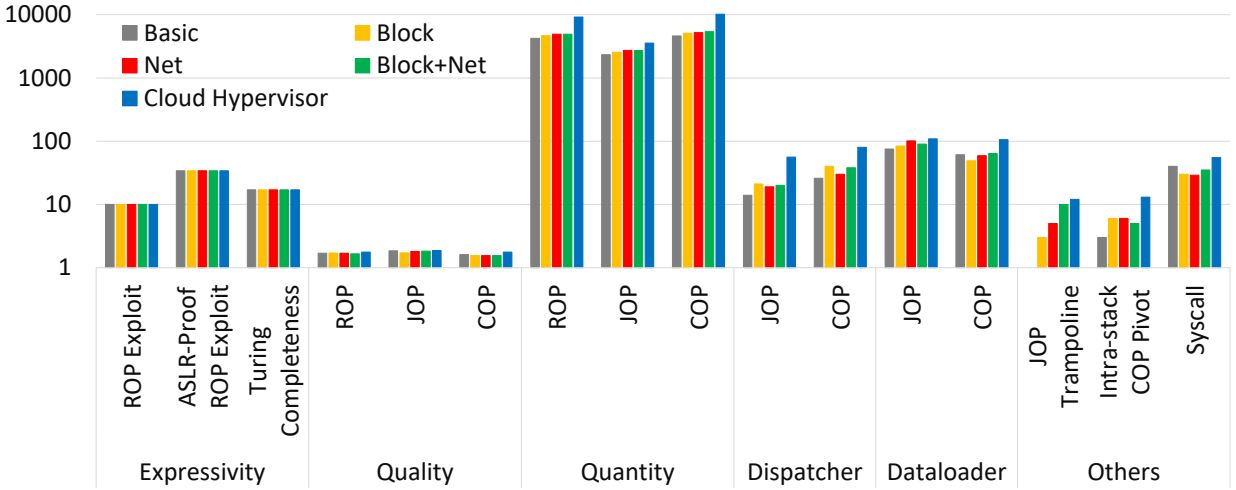


Figure 14.3: Gadget set analysis.

The figure in Figure 14.3 shows that none of the Redwood VMMs increase the expressivity of the fundamental functional gadget. However, the quality and quantity of ROP, JOP, and COP gadgets increase with the increase in components in a Redwood VMM, exhibiting the highest values for all default features enabled in Cloud Hypervisor. It essentially suggests that fewer components help reducing gadget quality and quantity. Redwood also reduces special-purpose gadgets that load data for and dispatch JOP and COP gadgets, as well as JOP trampoline, COP intra-stack pivot, and system call gadgets, compared to the default

Cloud Hypervisor. In summary, Redwood's flexible configuration options can significantly reduce an attacker's freedom to exploit gadgets presented in the VMM.

Chapter 15

Conclusions

Cloud infrastructures are heavily dependent on virtualization technologies. In this dissertation, we focus on the security risks associated with widely used virtualization technologies and develop several techniques to mitigate them, preserving existing performance.

The device drivers are very crucial yet erroneous components in an OS. While Xen supports moving device drivers to a separate VM to prevent device-related vulnerabilities from affecting the whole virtualization infrastructure, the OSs in those VMs have considerably bigger attack surfaces as they are general-purpose and fully featured. In contrast, using lightweight OSs like unikernels for the driver domain can reduce the attack surface significantly. Unfortunately, no existent unikernel works as a driver domain. Though rumprun unikernel benefits from NetBSD's rich driver base, it lacks backend drivers, the crucial component of any driver domain, implemented for Xen PV I/O devices. Rumprun lacks work queues, full Xen-tools support, and the ability to execute scripts. As these challenges did not receive adequate attention from prior researchers, this dissertation address them.

At the same time, existing VMMs are designed to be generic and fully featured so that one VMM can support VMs with a wide range of configurations. Their rigid design does not allow specializing VMMs based on the need of VMs, creating a large attack surface for each instance of a VMM. While VMMs provide isolation between VMs, there is no intra-VMM isolation for existing solutions to limit attackers' freedom to exploit bugs in one component and leverage that to steal data from or run arbitrary code on different

components. Since the vulnerability in a VMM can negate the security benefit of VMMs, it is crucial to bring changes to traditional VMM design, allowing per-VM specialization and intra-VMM compartmentalization. Therefore, this dissertation discusses new VMM principles addressing the mentioned issues and corresponding designs to implement them.

Performance acceleration architecture, such as DPDK and SPDK, are often incorporated with VMM and VM to improve I/O performance. However, there are potential security issues with such architectures, as they involve connecting multiple VM to and storing these VMs' sensitive information on the same userspace application. Any compromise in such software may compromise the security of all connected VMs. This dissertation is the first to address this issue and proposes security measures to prevent possible future attacks.

15.1 Contributions Revisited

15.1.1 Kite: Unikernelized Storage Domain

This dissertation presents a unikernelized storage driver domain. We leveraged rumprun, a lightweight and minimalistic unikernel. We imported the physical storage driver from NetBSD and introduced a Xen PV storage backend driver and a configuration application.

We evaluate the performance of the Kite storage domain with a rich set of well-known benchmarking tools. We also measure security and deployment properties and compare them with existing Linux-based OSs. Our evaluation shows that our Kite storage domains provide competitive performance to Linux-based storage driver domains. The security evaluation shows Kite has reduced attack surface, resulting from a minimal number of gadgets, smaller image sizes, and fewer system calls, which helps mitigate known vulnerabilities. Kite also exhibits a faster boot time.

15.1.2 Kite: Unikernelized Network Domain

This dissertation also presents a unikernelized network driver domain, which is also based on rumprun. Like storage drivers, we leverage NIC drivers from NetBSD. We design and implement the Xen PV network backend driver and a network bridging application.

Our performance evaluation on the Kite network domain involves running well-known real-life network applications and measuring their performance with micro- and macro-benchmarks. The results show that Kite is as performant as Linux-based network domains. Kite network domains are as secured as storage domains.

15.1.3 Redwood: Flexible Secure VMM

In this dissertation, we also present Redwood - a flexible and compartmentalized VMM based on new design principles to secure cloud infrastructure, preserving performance benefits. A flexible VMM model helps to reduce the attack surface significantly. Intra-VMM compartmentalization mechanisms provide resilience against vulnerability exploitation in VMMs.

We have implemented Redwood's prototype VMM, written in Rust, which has 18 features to choose from when building a VMM for any particular VM. It also separates 13 I/O devices into distinct compartments with the help of hardware features for memory protection. Our security evaluation reveals that Redwood can protect against many severe CVEs without losing performance or compatibility with cloud OSs and reduces attack surface up to 50%. Moreover, we run a rich set of benchmarks to assess the performance of different compartmentalized I/O devices in Redwood and compare them with another VMM called Cloud Hypervisor. The results show Redwood's compartmentalization does not incur any noticeable performance penalty. Redwood also enables support for running unikernels.

15.1.4 Vhost User Compartmentalization in DPDK

This dissertation proposes compartmentalizing sensitive VM information inside DPDK. We isolated each VM's memory mapping at the vhost-user interface in OVS-DPDK with the help of hardware-based memory protection keys. Our proposed model enables access to a VMs memory from the DPDK side only when necessary and keeps it disabled the rest of the time. Therefore, an attacker cannot arbitrarily access any VMs' memory from DPDK.

We do a security analysis and performance evaluation on DPDK with our proposed changes. The analysis shows modified DPDK's resilience to known CVEs. The enforced isolation does not incur any performance penalty, measured with popular benchmarking applications.

15.2 Perspective on Dissertation Contributions

Unikernelization approaches have been gaining popularity in recent years due to their ability to significantly reduce the size of virtual machines and enhance isolation between applications. Unikernels are therefore a great candidate for running cloud workloads. Prior approaches have mostly focused on unikernelizing applications, but Kite takes a different approach by unikernelizing driver domains.

Traditionally, full-fledged operating systems like Linux are capable of running multiple applications, scripts, physical device drivers, PV backends, and supporting rich workqueues, which allows them to run driver domains. However, until now, no unikernel has supported such infrastructure. Kite introduces techniques to overcome these limitations and run driver domains in a unikernel. Kite's approach to unikernelizing driver domains brings the benefits of unikernels in the context of disaggregating the hypervisor and reducing the presence of unwanted system calls while maintaining the high level of security and performance that

unikernels are known for.

Specialization is critical when developing software in a stack to meet specific requirements. Although there are numerous existing VMM approaches designed to run VMs for particular workloads, there is no general VMM approach that offers high flexibility enabling per-VM specialization. However, the Redwood system breaks down the general VMM architecture into fine-grained components and proposes highly customized VMMs that keep all existing features available for VMs. By utilizing Redwood's devised techniques, it is possible to achieve flexible specialization for general VMMs.

Many hypervisor disaggregation techniques aim to increase isolation between multiple hypervisor components by running them in separate VMs. This approach reduces the overall attack surface of the system. However, when it comes to isolating software components within the same process, intra-process compartmentalization techniques are more commonly used. These techniques usually focus on isolating different libraries from each other. However, Redwood breaks down the general VMM architecture into fine-grained components and proposes highly customized VMMs that retain all existing features available for VMs. By utilizing Redwood's techniques, we demonstrate how virtualization infrastructures can improve cloud security without compromising performance. In general, Redwood's six design principles for improving security in virtualization systems can be applied to both existing and future VMMs.

This dissertation reveals that acceleration architectures, such as DPDK, deployed to improve performance and isolation in virtualization infrastructure, can introduce security concerns. Our proposed solution for compartmentalizing sensitive VM information at the DPDK side demonstrates how hardware-based isolation techniques can be used to protect VMs in the event of an attack on the acceleration architecture in the cloud. Similar approaches can be used in other acceleration frameworks that are associated with VMs.

The central theme of the dissertation is improving the security of OS and virtualization software by isolation, i.e., by compartmentalizing (OS and virtualization) software into separate domains, vulnerabilities in one domain cannot be exploited to compromise other domains. This security principle can also be applied to other software systems, both infrastructure software as well as non-infrastructure software.

15.3 Limitations and Future Work

15.3.1 Ensuring Protection Key's Integrity in Redwood

Redwood's proposed compartmentalization can be implemented with different techniques, such as protection keys, memory tagging [1], hardware capability [136], etc., to isolate VMM components. Each technique has its own advantages and disadvantages. The Redwood prototype uses Intel MPK [67] because of its support in popular x86-based architecture and low-performance overhead. However, MPK-based memory protection is not perfect. The prototype assigns one distinct pkey per compartment, where compartments are mutually distrusting. It uses pkey to change access permission to the memory of a compartment as per invocation. The permission changes occure at a few distinct locations using a few lines of trusted code. However, if attackers can manage to inject or manipulate code to change pkey permission, they can compromise the compartmentalization. Therefore, it is crucial to ensure that the pkey permission changes only happen at designated locations using legitimate codes, guaranteeing integrity in VMM's code control flow.

Even if not for VMM compartmentalization, researchers have addressed this problem in different contexts. For instance, ERIM's [131] security model compartmentalizes applications into a trusted and untrusted domain. ERIM assumes the trusted domain is not exploitable.

ERIM uses call gates and safe instructions to switch between the two domains. Any other instructions are considered unsafe and are either replaced by functionally equivalent sequences or monitored at runtime. At startup, ERIM verifies the absence of unsafe instructions in the protected application and terminates the program if the monitor finds an unsafe instruction being executed. Hodor [77] uses a trusted application loader to partition the application into trusted and untrusted libraries. The pkey-based sandbox monitors the application at runtime to prevent the abuse of unsafe instructions. When the application marks a page as executable, the trusted loader first scans the page for unsafe instructions and triggers a fault if any are found. Hodor requires kernel modification to either put hardware breakpoints on the unsafe instructions or single-step through the page to ensure that the kernel vets all unsafe instructions. If a hardware breakpoint is triggered, the program execution is terminated. However, their technique comes with execution overhead because of monitoring and has some security and usability challenges [132].

We propose comprehensive research for finding an appropriate method that ensures attackers cannot change pkeys permission on a compartment even if they gain control of some part of VMM. Techniques such as execution monitoring, anomaly detection [122], control flow integrity, or control flow execution [30] can be taken into consideration to achieve such a guarantee. It would also be a potential problem to understand incurred performance penalties in VMMs because of the introduced security mechanisms that ensure pkeys safety.

15.3.2 Automating Compartmentalization

Redwood's proposed principle and design for compartmentalization help limit an adversary's freedom to exploit bugs in one VMM component to manipulate an attack in another component. We exercised a protection key-based memory isolation technique for compart-

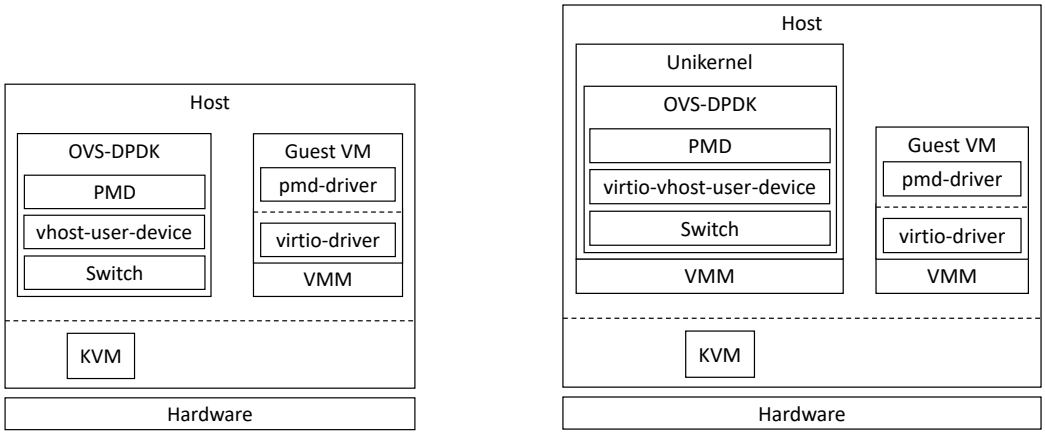
mentalizing I/O devices in Redwood's prototype. We manually identified what memories are allocated and used for those devices and set the pkey to those memory locations accordingly, which requires a careful understanding of VMM implementation. However, there is always a chance that the developer may miss setting a pkey to an expected memory location.

Therefore, we propose automating the compartmentalization process to identify all memory locations associated with a VMM component after taking a component's identity as an input. One way to achieve such automation is by making the compiler analyze the intermediate representation of a VMM code and annotate all memory locations of a component with a pkey. Another way would be developing a script that will automatically edit the VMM's code for isolating components. We believe figuring out the best method for automatic VMM compartmentalization would be an interesting future work.

15.3.3 Unikernelized Vhost User Devices

Offloading the implementation of virtio devices, referred to as vhost-user devices, to the host's userspace increases isolation and reduces the attack surface of a VMM. Acceleration technologies such as DPDK and SPDK, as shown in Figure 15.1a, use this approach to connect to guest VMs. However, running the acceleration application directly on the host increases the host's attack surface, as each application has its own vulnerabilities. To address this, running the entire acceleration application inside a guest VM provides better isolation between the host and the vhost-user device. The recent introduction of the virtio-vhost-user device [13] enables a guest VM to act as a vhost-user device, allowing applications such as DPDK and SPDK to run inside VMs and provide vhost devices to other VMs.

The virio-vhost-user implementation involves running OVS-DPDK inside a full-fledged Linux VM. As discussed in this dissertation before, a full-fledged Linux suffers from a large attack



(a) Regular vhost-user device. (b) Unikernelized virtio-vhost-user device.

Figure 15.1: Unikernelized acceleration architectures with virtio-vhost-user devices can provide VM-level isolation to reduce the attack surface.

surface. On the other hand, a VM with a virtio-vhost-user device does not need to be generic, as it will mostly run the OVS-DPDK application. Therefore, we propose running virtio-vhost-user devices as unikernels to reduce attack surface significantly, like Kite driver domains, as opposed to Linux. Figure 15.1 shows the current OVS-DPK vhost-user architecture (a) along with the proposed architecture with unikernel (b) for the same setup. It may need some investigation to fit the application (the one that will run with the virtio-vhost-user device) and its associated scripts or daemons in the unikernel. The introduced indirection can cause some performance hits and may require some optimization to minimize that, which can be another interesting research problem.

15.3.4 Unikernelized Virtio Driver Domain for Xen and KVM

The Virtio specification provides a higher number of PV devices compared to those offered by Xen device models. Furthermore, Virtio is widely adopted across various hypervisors in virtualization environments, making its integration with Xen a crucial and ongoing effort. Despite this, Xen's driver domain model is a distinctive feature not found in other

hypervisors. Driver domains help to isolate critical components and reduce the scope of vulnerability. As a result, the implementation of Virtio devices in Xen and the driver domain in KVM can enable the virtio driver domain for both hypervisors, which is a very desired feature.

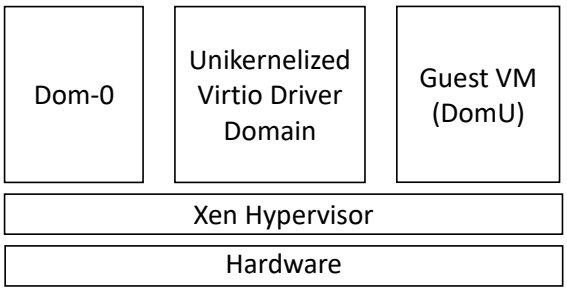


Figure 15.2: Architecture for unikernelized virtio driver domain on Xen.

On the other hand, the Kite driver domain shows how unikernelizing critical device driver domains can reduce attack surface significantly without losing any performance benefit. Therefore, this dissertation proposes the unikernelization of the driver domain capable of running virtio devices on Xen and KVM as future work, as depicted in Figure 15.2. We expect similar security and performance benefits from the virtio driver domain that Kite offers. Therefore, a Xen- and Kvm-based virtualization system can enjoy more PV devices with enhanced security, keeping a minimal attack surface.

15.3.5 Rumprun PVH

As shown in Figure 15.3, Xen offers multiple virtualization modes. For the work presented in this dissertation, we leveraged rumprun with HVM (or PVHVM to be specific) support from LibrettOS [103]. However, the last addition to this list is PVH, which is a lightweight virtualization mode that does not require any QEMU emulation and benefits from hardware-accelerated virtualization. The PVH support in rumprun is yet to be implemented, which

can be another direction for future work.

Poor Performance
Scope for Improvement
Optimal Performance

PV = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized
HA = Hardware Accelerated

x86 Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Boot Path	Privileged Instructions, Page Tables	QEMU Used
HVM / Fully Virtualized	HVM		VS	VS	VS	VH	Yes
HVM + PV drivers	HVM	PV Drivers Installed	PV	VS	VS	VH	Yes
PVHVM	HVM	PVHVM Capable Guest	PV	PV	VS	VH	Yes
PVH	PVH	PVH Capable Guest	PV	HA	PV	VH	No
PV	PV		PV	PV	PV	PV	No
ARM							
N/A	N/A		PV	VH	PV	VH	No

Figure 15.3: Overview of the various virtualization modes implemented in Xen. (Courtesy: Xen Project [29])

There are similarities between PVHVM and PVH mode. Both use hardware virtualization, such as Intel VT or AMD-V extensions of the host CPU, for virtualizing. For I/O device virtualization, such as network and storage, both PVHVM and PVH rely on PV device drivers. Hardware virtualization features are used for trapping privileged instructions and manipulating page tables by both modes.

PVHVM uses Software Acceleration, such as Local APIC, Posted Interrupts, Viridian (Hyper-V) enlightenments, and makes use of guest PV interfaces because they are faster. In contrast, PVH leverages hardware acceleration support instead of the PV interface. Unlike HVM guests, PVH guests do not require QEMU to emulate devices, which makes the PVH guests lighter than PVHVM.

We expect that enabling rumprun with the PVH mode would make it lightweight and faster than it is now with the PVHVM mode. However, there are challenges in doing that, such as adopting PVH for rumprun, which we did not need for PVHVM. Moreover, PVH integration is still a work-in-progress, and several features are yet to be added to Xen. For instance, PCI passthrough support is still absent, but it is essential for driver domain implementation. We anticipate that these missing features will be supported soon and propose to upgrade our rumprun-based Kite driver domains.

Bibliography

[1] [n. d.]. Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Accessed: 2022-01-08.

[2] [n. d.]. Open Virtual Machine Firmware. http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt. Accessed: 2022-01-08.

[3] [n. d.]. Rust Hypervisor Firmware. https://github.com/cloud-hypervisor/rust-hypervisor-firmware. Accessed: 2022-01-08.

[4] [n. d.]. vDPA - virtio Data Path Acceleration . https://vdpa-dev.gitlab.io/. Accessed: 2022-01-08.

[5] [n. d.]. VFIO - "Virtual Function I/O" . https://docs.kernel.org/driver-api/vfio.html. Accessed: 2022-01-08.

[6] 2015. Dom0. https://wiki.xenproject.org/wiki/Dom0 Online, accessed 02/01/2023.

[7] 2015. The VENOM vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[8] 2016. Docker Acquires Unikernel Systems to Extend the Breadth of the Docker Platform. https://www.docker.com/docker-news-and-press/docker-acquires-unikernel-systems-extend-breadth-docker-platform.

[9] 2016. Elastic Network Adapter – High Performance Network Interface for Amazon EC2. https://aws.amazon.com/blogs/aws/

elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/ Online, accessed 02/01/2023.

[10] 2018. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ Online, accessed 09/15/2018.

[11] 2018. Ubuntu 18.04.6 LTS (Bionic Beaver). https://releases.ubuntu.com/18.04/ Online, accessed 02/01/2023.

[12] 2019. CVE-2019-18960. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18960.

[13] 2019. Virtual I/O Device (VIRTIO) Version 1.1. https://uarif1.github.io/vvu/virtio-v1.1-cs01 Online, accessed 02/02/2023.

[14] 2020. CVE-2020-14377. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14377.

[15] 2020. dd – convert and copy a file. https://man7.org/linux/man-pages/man1/dd.1.html.

[16] 2020. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[17] 2020. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[18] 2020. Memcached. http://memcached.org/.

[19] 2020. mongoDB. The database for modern applications. https://www.mongodb.com/ Online, accessed 05/26/2020.

[20] 2020. MySQL. https://www.mysql.com/.

[21] 2020. Netperf Manual. http://www.cs.kent.edu/~farrell/dist/ref/Netperf.html.

[22] 2020. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark/.

[23] 2020. Redis. https://redis.io/.

[24] 2020. Ropper. https://github.com/sashs/Ropper Online, accessed 05/26/2020.

[25] 2020. Ropper. https://github.com/sashs/Ropper.

[26] 2020. SysBench Manual. https://man7.org/linux/man-pages/man1/dd.1.html.

[27] 2020. SYSSTAT Utilities. http://sebastien.godard.pagesperso-orange.fr/.

[28] 2020. The NetBSD Project. https://netbsd.org.

[29] 2020. Xen Project. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview Online, accessed 05/26/2020.

[30] 2021. Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/.

[31] 2021. Nuttcp Welcome Page. https://www.nuttcp.net.

[32] 2021. Xen application CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+crafted+application.

[33] 2021. Xen application CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+shell.

[34] 2022. An alternative to nested virtualization. https://research.ibm.com/blog/nested-virtualization-free-turtles.

[35] 2022. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[36] 2022. cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[37] 2022. GadgetSetAnalyzer. https://github.com/michaelbrownuc/GadgetSetAnalyzer.

[38] 2022. kvmtool. https://github.com/kvmtool/kvmtool.

[39] 2023. Amazon Elastic Compute Cloud Documentation. https://docs.aws.amazon.com/ec2/index.html Online, accessed 02/01/2023.

[40] 2023. Common Vulnerability Scoring System SIG. https://www.first.org/cvss/ Online, accessed 02/01/2023.

[41] 2023. CVE. https://cve.mitre.org/ Online, accessed 02/01/2023.

[42] 2023. CVE Details. https://www.cvedetails.com/ Online, accessed 02/01/2023.

[43] 2023. What Is a Cloud Server? https://www.rackspace.com/library/what-is-a-cloud-server Online, accessed 02/01/2023.

[44] IEEE Std 802.11 a. 1999. Wireless LAN medium access control (MAC) and physical layer (PHY) specification: high-speed physical layer in the 5GHz band. (1999).

[45] Cloud Hypervisor a Series of LF Projects LLC. 2022. Cloud Hypervisor. https://www.cloudhypervisor.org/.

[46] Md Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In *26th ACM Conference on Computer and Communications Security (CCS)*. 1803–1820.

[47] AMD, Inc. [n. d.]. AMD I/O Virtualization Technology (IOMMU) Specification. http://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf.

[48] T. E. Anderson. 1992. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. 92–94. https://doi.org/10.1109/WWOS.1992.275682

[49] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. 164–177. https://doi.org/10.1145/945445.945462

[50] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-Level Access to Privileged CPU Features *(OSDI'12)*. USENIX Association, USA, 335–348.

[51] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization

[52] Timo Bingmann. [n. d.]. pmbw - Parallel Memory Bandwidth Benchmark / Measurement. https://panthema.net/2013/pmbw/. Accessed: 2022-01-04.

[53] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[54] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, USA, 309–322.

[55] Manel Bourguiba, Kamel Haddadou, Ines KORBI, and Guy Pujolle. 2014. Improving Network I/O Virtualization for Cloud Computing. *Parallel and Distributed Systems, IEEE Transactions on* 25 (03 2014), 673–681. https://doi.org/10.1109/TPDS.2013.29

[56] Manel Bourguiba, Kamel Haddadou, and Guy Pujolle. 2012. Packet Aggregation Based Network I/O Virtualization for Cloud Computing. *Computer Communications* 35 (02 2012), 309–319. https://doi.org/10.1016/j.comcom.2011.10.002

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

[58] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized through Software Debloating *(CSET'19)*. USENIX Association, USA, 5.

[59] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.

[60] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[61] Shakeel Butt, H Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. 2012. Self-service cloud computing. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 253–264.

[62] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. 2022. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *USENIX Security Symposium*.

[63] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. 56–71. https://doi.org/10.1109/SP.2016.12

[64] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-Oriented Attacks with Existing and Potential Defense Approaches. *ACM Trans. Priv. Secur.* 24, 4, Article 26 (sep 2021), 36 pages. https://doi.org/10.1145/3462699

[65] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 189–202. https://doi.org/10.1145/2043556.2043575

[66] QEMU contributors. 2022. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[67] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/.

[68] Intel Corp. 2018. Intel Clear Containers. https://clearlinux.org/documentation/clear-containers.

[69] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf.

[70] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (London, UK) *(ESSoS 2016)*. Springer-Verlag, Berlin, Heidelberg, 155–172. https://doi.org/10.1007/978-3-319-30806-7_10

[71] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (London, UK) *(ESSoS 2016)*. Springer-Verlag, Berlin, Heidelberg, 155–172. https://doi.org/10.1007/978-3-319-30806-7_10

[72] Cloud Native Computing Foundation. 2020. Production-Grade Container Orchestration. https://kubernetes.io/.

[73] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association

for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728

[74] Tom Goethals, Merlijn Sebrechts, Ankita Atrey, Bruno Volckaert, and Filip Turck. 2018. Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications. 1–8. https://doi.org/10.1109/SC2.2018.00008

[75] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. 2020. IskiOS: Intra-kernel Isolation and Security using Memory Protection Keys. (2020).

[76] Irfan Habib. 2008. Virtualization with KVM. *Linux Journal* 2008, 166 (2008), 8.

[77] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 489–503.

[78] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327

[79] Intel Corporation. [n. d.]. Intel's Virtualization for Directed I/O. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[80] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical

Memory Corruption Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1612–1626. https://doi.org/10.1145/3460120.3485376

[81] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. 2022. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *2022 IEEE Symposium on Security and Privacy (SP)*. 650–665. https://doi.org/10.1109/SP46214.2022.9833650

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

[83] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating system support for virtual machines. In *ATEC '03: Proceedings of the 2003 USENIX Annual Technical Conference*. 71–84.

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

[85] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217

[86] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the 15h European Conference on Computer*

Systems (Heraklion, Greece) *(EuroSys '20)*. Article 11, 15 pages. https://doi.org/10.1145/3342195.3387526

[87] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*.

[88] David Law. 2019. IEEE Standard for Ethernet-Amendment 1: Physical Layer Specification and Management Parameters for 2.5 Gb/s and 5 Gb/s Operation over Backplane. *IEEE Std 802.3 cb-2018 (Amendment to IEEE Std 802.3-2018)* (2019).

[89] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86 *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1441–1454. https://doi.org/10.1145/3243734.3243748

[90] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. https://doi.org/10.1145/3503222.3507759

[91] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[92] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance *(OSDI'16)*. USENIX Association, USA, 49–64.

[93] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[94] ARM Ltd. 2009. Building a Secure System using TrustZone Technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531?token=.

[95] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 461–472.

[96] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 461–472.

[97] Anil Madhavapeddy and David J Scott. 2013. Unikernels: Rise of the virtual library operating system. *Queue* 11, 11 (2013), 30–44.

[98] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 218–233. https://doi.org/10.1145/3132747.3132763

[99] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function

Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI '14)*. 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[100] A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. 2022. Kite: Lightweight Critical Service Domains *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 384–401. https://doi.org/10.1145/3492321.3519586

[101] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[102] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. 2008. Improving Xen security through disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*. 151–160.

[103] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. 114–128. https://doi.org/10.1145/3381052.3381316

[104] Ron Oglesby and Scott Herold. 2005. *VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide series)*.

[105] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[106] Pierre Olivier, AKM Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. 2019. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 85–96.

[107] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 241–254.

[108] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. Nojitsu: Locking down javascript engines. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society.

[109] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. KR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 420–436. https://doi.org/10.1145/3064176.3064216

[110] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 291–304. https://doi.org/10.1145/1961295.1950399

[111] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the 17th Workshop on Hot Topics*

in Operating Systems (Bertinoro, Italy) *(HotOS'19)*. 7–13. https://doi.org/10.1145/3317550.3321445

[112] John Scott Robin and Cynthia E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*. 129–144.

[113] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[114] Joanna Rutkowska and Rafal Wojtczuk. 2010. Qubes OS architecture. Invisible Things Lab Tech Rep.

[115] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558. https://doi.org/10.1145/3445814.3446731

[116] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, 671–688. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg

[117] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, USA, 1.

[118] Amazon Web Services. 2022. Firecracker: Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/.

[119] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[120] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *NDSS*.

[121] Sushrut Shirole. 2014. Performance Optimizations for Isolated Driver Domains.

[122] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-Span Program Behavior Modeling and Attack Detection. *ACM Trans. Priv. Secur.* 20, 4, Article 12 (sep 2017), 28 pages. https://doi.org/10.1145/3105761

[123] SPDK Contributors. [n. d.]. Storage Performance Development Kit (SPDK). http://spdk.io/.

[124] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3381052.3381326

[125] Sysbench Contributors. [n. d.]. SysBench 1.0: A System Performance Benchmark. http://sysbench.sourceforge.net/.

[126] V. Tarasov, E. Zadok, and S. Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.* 41 (2016).

[127] The Apache Software Foundation. [n. d.]. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[128] The Linux Foundation. [n. d.]. Data Plane Development Kit (DPDK). http://dpdk.org/.

[129] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, USA) *(ATC'17)*. 645–658. http://dl.acm.org/citation.cfm?id=3154690.3154752

[130] Michael S Tsirkin, C Huck, and P Moll. 2018. Virtual I/O Device (VIRTIO) Version 1.1. *OASIS Committee: Burlington, MA, USA* (2018).

[131] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1221–1238.

[132] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[133] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993.

Efficient Software-Based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (dec 1993), 203–216. https://doi.org/10.1145/173668.168635

[134] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security* (Heraklion, Greece) *(EuroSec '20)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3380786.3391398

[135] Jon Watson. 2008. Virtualbox: bits and bytes masquerading as machines. *Linux Journal* 2008, 166 (2008), 1.

[136] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

[137] D. Williams and R. Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[138] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box *(HotCloud'16)*. USENIX Association, USA, 71–76.

[139] Bruno Xavier, Tiago Ferreto, and Luis Jersak. 2016. Time provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. In *Proceedings of the 16th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '16). 277–280.

[140] Xen Project. 2013. Driver Domain. https://wiki.xenproject.org/wiki/Driver_Domain.

[141] Xen Project. 2018. Grant Table. https://wiki.xen.org/wiki/Grant_Table.

[142] Lingfang Zeng, Yang Wang, Dan Feng, and Kenneth Kent. 2015. XCollOpts: A Novel Improvement of Network Virtualization in Xen for I/O-Latency Sensitive Applications on Multicores. *IEEE Transactions on Network and Service Management* 12 (06 2015), 1–1. https://doi.org/10.1109/TNSM.2015.2432066

[143] Mingwei Zhang, Ravi Sahita, and Daiping Liu. 2018. eXecutable-Only Memory-Switch (XOM-Switch).

[144] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*.