Kite: Lightweight Critical Service Domains

A K M Fazla Mehrab
mehrab@vt.edu
Virginia Tech
Blacksburg, VA, USA

Ruslan Nikolaev
rnikola@psu.edu
The Pennsylvania State University
University Park, PA, USA

Binoy Ravindran
binoy@vt.edu
Virginia Tech
Blacksburg, VA, USA

Abstract

Converged multi-level secure (MLS) systems, such as Qubes OS or SecureView, heavily rely on virtualization and service virtual machines (VMs). Traditionally, driver domains – isolated VMs that run device drivers – and daemon VMs use full-blown general-purpose OSs. It seems that specialized lightweight OSs, known as unikernels, would be a better fit for those. Surprisingly, to this day, driver domains can only be built from Linux. We discuss how unikernels can be beneficial in this context – they improve security and isolation, reduce memory overheads, and simplify software configuration and deployment. We specifically propose to use unikernels that borrow device drivers from existing general-purpose OSs.

We present *Kite* which implements network and storage unikernel-based VMs and serve two essential classes of devices. We compare our approach against Linux using a number of typical micro- and macrobenchmarks used for networking and storage. Our approach achieves performance similar to that of Linux. However, we demonstrate that the number of system calls and ROP gadgets can be greatly reduced with our approach compared to Linux. We also demonstrate that our approach has resilience to an array of CVEs (e.g., CVE-2021-35039, CVE-2016-4963, and CVE-2013-2072), smaller image size, and improved startup time. Finally, unikernelizing is doable for the remaining (non-driver) service VMs as evidenced by our unikernelized DHCP server.

CCS Concepts: • **Security and privacy → Operating systems security**; **Virtualization and security**.

Keywords: Hypervisor, Virtual Machine, Unikernel, Xen

ACM Reference Format:
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), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519586

EuroSys '22, April 5–8, 2022, RENNES, France

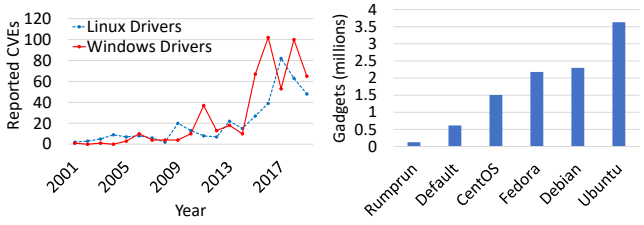
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519586

1 Introduction

Converged multi-level secure (MLS) systems [24, 27, 32, 35, 58, 65] are OSs which allow users to interact with applications easily in a single environment while providing better levels of security than typical OSs [42]. More specifically, an OS can use hardware virtualization as an extra protection layer against malicious actors. Based on the MLS OS's threat model and design goals, malicious actors can include *insiders* (users) as well as *outsiders* (attackers), and additional sensors can run in a privileged virtual machine (VM) to monitor abnormal behavior of both insiders and outsiders. An MLS OS can be deployed both locally for desktop users (e.g., Qubes OS [65]) and on a cloud, such as Amazon EC2, for enterprise users, especially when using newer Amazon bare-metal instances which expose physical network and storage device interfaces.

Typically, MLS OSs consist of a hypervisor (e.g., Xen [26] in Qubes OS [65] or SecureView [24]) and unprivileged VMs that run applications and system services. The idea is to run an application or a group of applications in isolated VMs, while providing the illusion of a uniform system to users. Application VMs can typically be of any type, e.g., Windows- or Linux-based. Special *service VMs* run daemons and drivers. Driver VMs use hypervisor's capabilities of PCI passthrough [82] to access the corresponding device, and essentially run a physical driver (for storage or networking) in an isolated VM, known as a *driver domain* in Xen, for more effective load balancing and enhanced security. (See Section 2 for an overview of Xen, its I/O drivers, and driver domains.) Isolating drivers in separate VMs is especially important as the number of common vulnerabilities and exposures (CVE) for drivers continues to surge across different OSs (Figure 1a).

A downside of service VMs is that they are relatively heavy-weight as they usually run a general-purpose, full-fledged OS such as Linux. Such VMs are cumbersome for deployment and upgrades. Linux is designed to support many subsystems (e.g., audio, video, USB, etc) 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. (Ubuntu Server 18.04's image is ≈1GB; the kernel alone, without any modules, is ≈50MB.) 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



(a) CVEs, https://cve.mitre.org/. **(b)** ROP gadget comparison.

Figure 1. CVEs and ROP gadgets.

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 to run a Linux-based OS. In Section 5, we show that **the number of required Linux system calls is as high as 171**.

One can wonder **what it will take** to run a truly lightweight domain, which is exactly the problem that we address in our paper. Linux driver domains require xen-utils [22], a collection of user-space tools to manage a Xen-virtualized system. Xen-utils are also used to establish connections between driver domains and other guest OSs. Xen-utils depend on many other libraries and tools, which in turn also have numerous dependencies (e.g., Python, etc). Note that only a tiny fraction of this code is *really* relevant to driver domains. However, xen-utils' current monolithic approach cannot be trivially modified to disaggregate unnecessary components. It was reported previously that bugs or improper use of Python can let an attacker gain unauthorized privileges in Xen [2]. Moreover, xen-tools' *libxl* can often be a source of numerous vulnerabilities [4], which cause unauthorized access to sensitive locations, denial of service attacks, etc. That said, libxl could have been potentially substituted with a much more minimalistic approach suitable for driver domains, which avoids many of these pitfalls. However, the feasibility of that was not explored in the past. Altogether, since we still use a general-purpose OS for driver domains, this enables an attacker to tailor application vulnerabilities. (Not all applications can be fully excluded from driver domains.) Furthermore, applications can even be crafted [18] by the attacker to expose critical vulnerabilities.

Although the Linux kernel can be shrunk (but only to a certain degree), we still need a functional user-space environment, which includes not just GLibC but also shells, scripts, interpreters, configuration and bootstrapping tools, etc. Since the system calls are the gateway from user-space to kernel space, the attackers often use them to exploit vulnerabilities in the kernel and userspace. The attackers use vulnerable/malicious applications that eventually use the system calls to perform privilege escalation, denial of service attacks, or leak sensitive data. The presence of the large number of system calls in Linux (≈300) contributes to a huge attack surface. Therefore, most of the reported CVEs benefit from them, which makes system call reduction an active research area [39, 40]. However, many system calls are tightly coupled with the Linux kernel (e.g., clone, init_module, modify_ldt, etc), which makes it impossible to get rid of them without distorting the kernel design.

In contrast, unikernels [28, 43, 46–48, 51–53, 61, 83] are lightweight OSs designed specifically for cloud systems and run atop a hypervisor in separate virtual machines. They are, by design, capable of running only a single application. In unikernels, a single application is statically compiled together with the minimum necessary kernel code and libraries to produce a single-address-space image. Such an approach reduces code and memory footprints, thereby reducing the attack surface. Additionally, since this strategy eliminates context switching, system calls now become ordinary function calls. Elimination of context switching overheads can yield performance benefits. However, unikernels are mostly designed for user applications in mind and are unsuitable for driver domains.

In this paper, we explore the use of the rumprun unikernel [43] for this purpose. The key feature of rumprun is that it is directly based on NetBSD's code [16], which potentially enables access to NetBSD's large collection of device drivers, even very specialized ones such as Amazon ENA. (See Section 2 for an overview of rumprun.) The entire rumprun OS image is ≈22MB. To get a general idea of the security properties (putting Xen-related vulnerabilities aside) of the rumprun unikernel and full-fledged OSs, we measured the corresponding number of return-oriented programming (ROP) gadgets [64, 67] for rumprun, default-configured (fairly minimal and almost no modules), CentOS 8, Fedora 05/2020, Ubuntu 18.04, and Debian 10.4 Linux kernels with their associated kernel modules. Figure 1b demonstrates that rumprun has a substantially smaller number of gadgets than any of the Linux configurations (even without taking into account user-space ROP gadgets). Assuming that NetBSD's code quality is on par with that of Linux, this also indicates potential for improved security more generally, as rumprun's attack surface is proportionally reduced compared to that of full-fledged OSs. Furthermore, rumprun can **reduce the number of system calls by a factor of 10x**, and many of the disabled system calls are known to have CVEs (Section 5).

However, out-of-the-box rumprun lacks many critical features that are necessary to make driver domains feasible. For example, network and storage I/O must be exposed to guest VMs via efficient paravirtualized (PV) *frontend* and *backend* drivers, which communicate with the corresponding physical device drivers in a driver domain. A recent work [58] extends rumprun for multicore systems and implements minimalistic support for Xen's hardware virtualization mode (HVM). However, rumprun still lacks critical features in HVM (e.g., xenbus [79] and xenstore [80]). Rumprun also

lacks backend drivers for storage and networking. (Unfortunately, NetBSD's Xen drivers, unlike other drivers, cannot be used in rumprun, and NetBSD itself is not known to support driver domains.) Finally, rumprun lacks even basic system orchestration tools, e.g., bridging or network address translation (NAT), and configuration scripts that would make driver domains feasible.

We overcome these challenges by designing and implementing *Kite*, a system of unikernelized service VMs. It features the HVM version of rumprun incorporated with missing Xen features, PV backend drivers, and applications to complement the missing Xen scripts. We propose a threaded model to overcome the lack of components, such as preemptive scheduler and work queues. Resultantly, Kite contains rumprun-based network and storage VMs – the two existent types of driver domains. To measure the effectiveness of our implementation, we run experiments using iPerf, Wget, Apache, MySQL, MongoDB, Memcached, Redis, etc (Section 5). Kite provides competitive performance to that of Linux-based driver domains, while retaining all the benefits of unikernels such as reduced number of gadgets, smaller image size, and faster boot time.

The paper's research contribution is unikernelized service VMs in MLS OSs. While past efforts explored increasing isolation in hypervisors and reducing their attack surface (Section 6), our work is the first to explore unikernels for isolation of non-hypervisor core components (device drivers, OS daemons). Although unikernels already power *application* VMs in clouds, our work is the first to demonstrate that *service* VMs in an MLS OS can also be built using unikernels.

Finally, Xen's configuration and orchestration infrastructure is fairly complex and requires rich user-space environment, e.g., scripting languages. We show that much simpler unikernels achieve the same driver domain functionality with smaller overheads. None of the prior unikernel works targeted the same problem, nor was it clear if driver domains were feasible in the unikernel environment at all (due to the heavy infrastructure in Linux). Not to mention that most unikernels simply lack physical device drivers since they are targeted for clouds.

2 Background

2.1 Xen HVM

Xen is a popular Type I hypervisor [44], often used by MLS OSs. Though initially Xen pioneered "paravirtualization" (PV) which required to modify OS kernels, later CPUs implemented hardware-assisted virtualization (VT-x, AMD-V, etc), and Xen widely adopted the hardware virtualization mode (HVM). HVM is preferred by MLS OS vendors due to an extra layer of hardware isolation. HVM also better supports IOMMU [25, 41], which is required for driver domains.

2.2 Xen I/O Drivers

Traditional I/O device emulation is inefficient due to substantial performance overheads [29]. Xen's original idea to use special I/O drivers for PV also carries over to HVM as long as the corresponding guest VM is *enlightened* about Xen's presence. Other hypervisors, e.g., VMware, Hyper-V, and KVM, implement similar faster I/O drivers.

In Figure 2, we show Xen's PV I/O driver architecture. PV drivers are typically divided into two parts: *frontend* and *backend*. A frontend driver runs in the guest VM, denoted as DomU. This driver provides an interface, similar to that of a corresponding physical driver (e.g., a network interface). The underlying PV driver implementation remains transparent to the applications in DomU. Therefore, the applications do not require any modifications to access the PV drivers. Frontends can be instantiated from the same or different DomUs.

A backend driver can run either in Dom0, the privileged administrative VM, or in a driver domain (see Section 2.3). It communicates with the physical device through the device interface provided by the physical device driver that runs in the same domain. Each backend driver is implemented such that it can connect to multiple frontends.

Backends and frontends connect and communicate with each other using Xen's protocol. Both ends store their critical information in the *xenstore* [80] database. *Xenbus* [79] reads from and writes to xenstore. Using xenbus, the two sides negotiate their capabilities and features. To monitor the other end's activity, each end can use xenbus to set up "watches."

For inter-VM notifications, Xen's *event channel* implements virtual interrupts. Xen shares memory between VMs through *grant tables*, which specify what memory pages are shared. The frontend allocates shared memory and initializes I/O ring buffers, which are used by both ends to transfer data. As Figure 2 shows, for each frontend instance, a corresponding backend instance is created.

2.2.1 Netfront and Netback. For networking, *netfront* and *netback* correspond to frontend and backend drivers. Netfront exposes a virtual network interface to the network stack of a guest OS. On the other side, netback forwards packets to and from the physical device driver. 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.

There can be multiple netback instances in the VM. To share a network interface controller (NIC), a *bridge* can be used to connect all netbacks with the NIC. The bridge routes packets between backends and the NIC, and across different backends.

2.2.2 Blkfront and Blkback. For secondary storage, *blkfront* and *blkback* correspond to frontend and backend. A

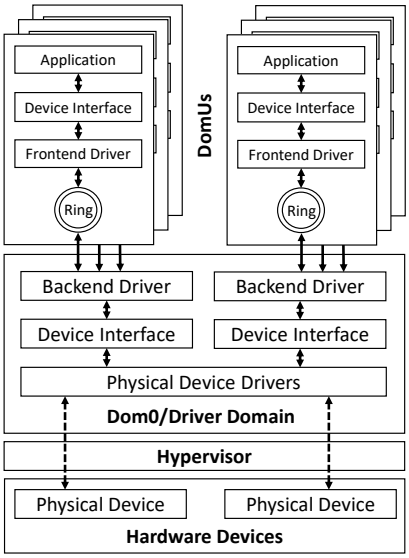


Figure 2. Xen's PV I/O driver model.

guest OS uses blkfront to issue block device operations such as read, write, etc. On the other side, blkback forwards these operations to the physical device driver. Using the I/O ring buffer, blkfront sends requests with the necessary information (sector number, size, etc) to blkback. Blkback performs the specified operation on the storage device and sends back a response. The data is transferred between two sides using shared memory via Xen's grant table.

2.3 Driver Domains

Xen's privileged (Dom0) VM traditionally runs device drivers and performs many critical system tasks. However, unprivileged guest VMs which run device drivers, known as *driver domains*, can offload Dom0. Driver domains also increase isolation and overall system security since potentially vulnerable/malicious drivers (or devices) are isolated from Dom0. Driver domains have direct access to the underlying hardware by using PCI passthrough [82]. In Xen, driver domains can be used for both networking and storage by running netback and blkback drivers, along with the corresponding physical device drivers, inside separate VMs rather than Dom0.

Note that Xen's split driver model is different from Xen's device virtualization (SR-IOV). Although VFs can be created for certain (but not all) NICs, they are less feasible for storage. Thus, dedicated driver domains are still often used to share the same piece of hardware across different VMs.

Because of the above-mentioned benefits, MLS OSs, such as Qubes OS [65] and SecureView [24], widely adopted driver domains. For stronger isolation, MLS OSs also require the I/O memory management unit (IOMMU) [25, 41] to be present. HVM is essential for full-fledged IOMMU support; it safely remaps interrupts and memory addresses to protect against both malicious (or faulty) devices and vulnerable (or buggy) device drivers. As a result, Dom0's attack surface reduces. Any driver malfunction or exploit will not directly affect Dom0, and Xen's administrative interface to other guest OSs remains uninterrupted. Though other hypervisors such as KVM [45] support faster I/O, they do not yet realize driver domains. (At the very least, they do not provide the same level of isolation.)

2.4 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 (Ethernet [49], Wireless LAN [23], etc) and storage (PATA, SATA, SCSI, NVMe) controllers must be supported. Because porting incurs non-trivial engineering effort, we need a unikernel that can reuse existing drivers.

Historically, the Flux OSKit [37] was the first to introduce the idea of constructing OSs with the components from multiple different OSs. NetBSD [16], 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.

Rumprun is a unikernel that leverages rump kernels such that it can potentially reuse any NetBSD device driver. Figure 3a shows the rumprun software stack, which consists of the platform-specific layer (Interface to 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. Kite 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 to that of 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 [47, 63] currently lack maturity and flexibility. In contrast, rumprun is stable, and rump kernels are upstreamed to NetBSD.

3 Design

In this section, we discuss the challenges for building driver domains and present a high-level overview of Kite.

3.1 Challenges

To retain all security benefits, we need to support IOMMU [25, 41], which confines erroneous DMA requests to a driver domain and prevents them from being propagated to other domains. Xen supports IOMMU fully only in the HVM mode. A recent work, LibrettOS [58], extends rumprun for multicore systems and partially adds Xen HVM support, which we leverage in our work. This rumprun variant still lacks *xenstore* and *xenbus* in HVM, which are required for Xen I/O drivers.

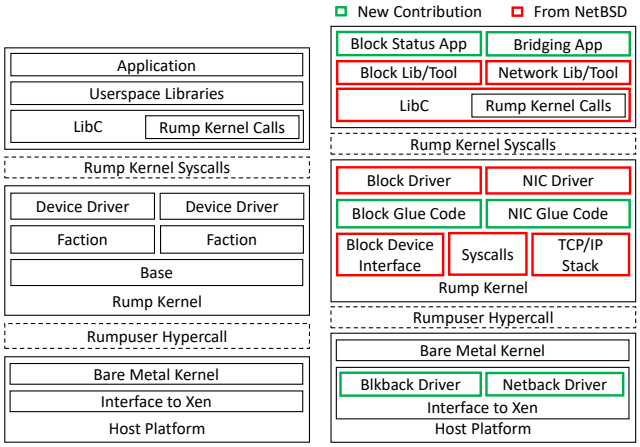
Another big hurdle is that all existing versions of rumprun only implement (non-optimized) frontend drivers and *completely lack* any backend drivers, let alone tools that would be necessary to configure backend drivers and driver domains. Unfortunately, NetBSD's Xen I/O drivers cannot be leveraged in rumprun unlike most other drivers. NetBSD is also not known to support driver domains at all.

Yet another hurdle is driver domain organization. There are several backend instances in driver domains. To link netbacks to a physical NIC, techniques such as bridging, routing, and network address translation (NAT) are used. Likewise, for storage, some device-specific information needs to be retrieved from and written to xenbus, which is later read by the corresponding blkback instances. Xen provides special domain initialization scripts and tools to perform these operations in Linux. They run as separate processes and are triggered on demand. However, a single-application unikernel environment cannot use this approach. Rumprun lacks the basic infrastructure to even run these scripts. Although NetBSD supports NAT and bridging, the corresponding tools must be ported and adapted as no one has ever used them in rumprun.

Unikernelization of driver domains requires more than implementing backend drivers, xenbus, xenstore, and tools. For example, netback must consistently maintain the data flow for both Rx and Tx ring buffers, which are handled asynchronously. In addition, orchestration tools must run concurrently as part of the unikernel application. Since rumprun uses a simple non-preemptive scheduler even for user code, both tools and netback must avoid ever monopolizing CPUs to allow seamless duplex communication without sacrificing performance or existing driver compatibility.

In our design, we largely retain the original functionality but use a more lightweight (single-process) infrastructure, written in C, which is specifically designed for our unikernel.

We consider the Xen hypervisor to be the only trusted component in a Xen-based virtualization stack. All VMs running on top of the hypervisor (Dom0, driver domains, and DomUs) are potentially vulnerable components and share the same threat model. Therefore, attacks applicable to guest VMs are also applicable to our driver domains.



(a) Basic rumprun stack on Xen. **(b)** Kite driver domain.

Figure 3. Driver domain adaptation to rumprun in Kite.

In Figure 3b, we show the architecture of our driver domains. Along with the basic rumprun components in different layers, we represent both domain-specific (network and storage) components, such as blkback and netback, in the same figure due to space limitations. The green rectangles denote the components that we developed from scratch, and the red rectangles present the components we adapted from NetBSD. Although each unikernel VM typically serves one device, our design can easily support many devices (e.g., several NICs for better I/O scaling) since Kite supports multiple cores.

3.2 Netback Driver

Two vital parts of the network driver domain are the netback driver and network device driver (e.g., ethernet driver). The network device driver, located at the rump kernel layer as shown in Figure 3b, provides a network interface (*IF*), which is used to transfer network packets between the netback driver and the outside world. The netback driver creates exactly one netback instance for every virtual network channel from the corresponding netfront instance in a guest VM. Each netback instance creates one virtual network interface (*VIF*). Our network domain connects all virtual and physical interfaces, i.e. *IF* and *VIF*s, using a network bridge, which is located at the application layer.

The *Interface to Xen* layer hosts the netback driver along with other Xen-related components such as xenbus and grant table interfaces. Even though netback is designed specifically for Xen, we have to separate it into platform-dependent and platform-independent layers to follow design principles and linking restrictions of rumprun. The upper layer of netback is responsible for communication with the network driver through rumprun's network stack and bridge. For any incoming packet (from the network stack) destined to a DomU, this

layer places packets into a memory buffer and forwards them to the bottom layer. The bottom layer places these memory buffers into a corresponding Xen I/O ring buffer and notifies 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.

This duplex communication happens through 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, we exploit multi-threading so that netback does not block any hypervisor-based event mechanisms and processes data fast. 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. Likewise, netback needs to respond to the network stack operations by using callback routines. Spending too much time in these routines delays subsequent operations because a response may issue an expensive hypercall. To minimize the response time, another thread is waken by the routines and performs the actual processing.

Linux runs a special *daemon* for starting services needed in the corresponding Xen driver domain. Along with other responsibilities, this service runs networking scripts that set up a bridge, NAT, etc. In our single-process unikernel environment, we create a *unified application* that waits for requests by DomUs, creates new netback instances, and connects netbacks to the physical NIC (i.e., using a bridge). This application cooperates with netback to avoid CPU monopolization.

3.3 Blkback Driver

In the same vein, blkback is instrumental for storage domains. If any VM seeks PV storage through blkfront, the storage domain negotiates with that VM and creates a blkback instance.

We divide blkback into platform-dependent and platform-independent layers similar to the netback driver. The bottom layer handles requests (i.e., transfers data blocks) from blkfront and sends responses using Xen I/O ring buffer. Depending on the type of the request (read, write, etc), the upper layer performs a corresponding operation on the storage device using the block device API. The upper layer also forwards a response from the device driver to the bottom layer.

Table 1. Lines of Code (LOC) for changes.

Component	Description	LOC
Blkback	Xen's storage backend driver	1904
Netback	Xen's network backend driver	2791
HVM extension	xenbus and xenstore support	1100
Configuration	network and storage applications	450
Utilities	ifconfig/brconfig changes	222
Daemon VM	OpenDHCP [60] as a daemon VM	16
Total:		**6483**

Each request consists of multiple block segments to perform a particular operation. A write request reads segments from the shared memory and writes them to the storage device. A read request is the opposite of that. Memory is shared through grant tables, which involve costly hypercalls, but our blkback driver adopts several optimizations described below.

Similar to netfront and netback, blkfront notifies blkback through the event channel. To prevent requests from being piled up and accelerate request processing, we run a separate thread for reading all pending requests and performing operations on the storage device. This thread wakes up only when there is a notification from the event channel.

When the physical device driver completes the requested operation, blkback sends a response to blkfront. Similar to Linux's blkback design, we handle operations and send responses asynchronously. Subsequent requests are not blocked by the current request. There are other Linux-specific optimizations we support to achieve good performance. We batch block device operations where segments from one or multiple requests are consecutive. This reduces the total number of block device operations and increases throughput. We use "persistent referencing" to avoid mapping and unmapping of the same memory location through grant table operations, which involve costly hypercalls. We retain a memory address and its grant reference, so that we can later reuse the existent mapping for a grant reference that is already saved.

Finally, a direct segment request contains a maximum of 11 segments' information since it is the maximum size a block ring can accommodate along with the ring indexes [21]. Therefore, *direct* segments are limited to 44KB, which is insufficient for high-speed NVMe SSDs. An indirect segment request contains grant references to pages, each containing 512 segments' information. Thus, we also support *indirect* segments for transferring up to 16MB per request.

4 Implementation

We use a recent version of rumprun [58] that already adds minimal HVM and multicore support. As Table 1 shows, most implementation effort relates to drivers. Configuration

applications, though small, support common scenarios; they cooperate with the driver code to avoid CPU monopolization.

4.1 Backend Invocation

A separate backend instance must be created for every frontend counterpart in DomU. A driver domain needs to know that one or more frontends are waiting for pairing. The backend driver sets a watch in xenstore [80], which is a database in the shared space between domains. The xenstored [78] daemon in Dom0 keeps this database updated with each domain's configuration and status information. Each domain has paths in xenstore where the corresponding information is stored. When a frontend instance attempts to connect to its backend instance, a new path containing the requesting frontends's information is created. Any change in driver domain paths is detected by the "watch" callback in the backend driver.

Kite backend driver spawns a dedicated *thread* in the very beginning of execution to handle all path changes. When the watch observes a change in the driver domain's path in xenstore, a corresponding callback function wakes this thread up. The thread queries xenbus [79] to check if there is any relevant change. For any unpaired frontend, the thread creates and initializes a new backend instance.

4.2 Netback Driver

Initialization. The first step is to update the xenstore database so that this netback's features are advertised to other domains.

Since the hypervisor has all machine memory mapped for all domains, it is faster to copy data across domains using the 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 feature. Therefore, in our Kite network driver domain, we implement this feature.

The communication between the frontend and backend parts happens through special I/O ring buffers, which are built on top of Xen's shared memory mechanism called grant tables [81]. Netfront and netback use two ring buffers: Tx and Rx. Netfront uses Tx to send data to netback, which eventually forwards this data to the network stack (e.g., bridge) through the netback virtual interface (VIF). Conversely, when netback's VIF receives any data from the stack, netback forwards it to netfront using Rx. To distinguish VIF instances from each other, each VIF is assigned a unique name.

Netfront allocates shared memory and initializes Tx and Rx. To access ring buffers, netback reads Tx and Rx grant references from xenstore and maps them to its address space. It keeps track of ring buffer accesses using producer-consumer indices. Notifications between netback and netfront happen asynchronously through event channels (virtual interrupts). Netback binds an event channel to a dedicated event handler.

Transmit. In DomUs, any data from the network stack goes to netfront. Netfront transmits packets to netback using Tx. Likewise, netback pushes received data to the corresponding VIF. The ring buffer consists of multiple slots that can be used for requests as well as responses. Tx transmits data from netfront through netback. After placing requests into the ring buffer, netfront issues a virtual interrupt. Netback's handler copies the unhandled requests and maps pages to its address space. The memory contents are then delivered to a VIF.

Netback sends responses to netfront on completed Tx operations by using the slots from already served requests. Once these slots are filled with responses, netback pushes them to the Tx ring buffer and notifies netfront (if necessary).

Receive. For the opposite direction, netback pushes the data from a VIF to Rx. Rx is not the inverse of Tx, but similar in some aspects. Like Tx, the Rx ring buffer consists of multiple slots that can be used for requests and responses.

When netfront sends Rx requests, netback retains them but cannot send back any data until it receives data from the VIF. When VIF data is ready, netback copies the data to the pages associated with the corresponding Rx requests using the grant table. To notify netfront, netback reuses the served request slots for responses and issues a virtual interrupt.

Multiple Threads. Rumprun lacks Linux's rich support for work queues. We use threads for faster Tx and Rx operations.

For faster response to the notifications coming from netfront, the notification handler at netback must act efficiently. If the handler responds to the requests using the shared ring buffers, subsequent notifications need to wait longer since the Xen hypercalls associated with shared memory manipulation are time expensive. It will hurt latency sensitive applications. Therefore, our handler only wakes up a dedicated thread, *pusher*, if not already awake. *Pusher* keeps reading Tx requests and copying corresponding data from the shared memory and sends them to the VIF through the network stack. If there are no pending Tx requests, the pusher thread goes to sleep.

If there is any incoming data at the VIF, a callback function in the netback driver is invoked. Similar to the pusher thread, a dedicated thread, *soft_start*, keeps copying and sending these data to the netfront instance using the Rx ring buffer and shared memory. This thread goes to sleep once there is no data or Rx requests left. The callback function only wakes up the *soft_start* thread if it is sleeping.

4.3 Network Application

Using a network bridge, we connect the VIF interfaces from netbacks to the physical NIC. To that end, we developed an application in our network driver domain. When this application is launched, it creates a bridge interface. Next, the application assigns an IP address to the physical interface; the physical interface works as a gateway for incoming

and outgoing packets across all VIFs. Then, the application watches for a new VIF to appear and adds the new interface to the bridge.

We ported the *ifconfig(8)* utility from NetBSD to initialize bridge interfaces and assign IP addresses. Along with that, we also ported the *brconfig(8)* utility from NetBSD, which is used for adding interfaces to a bridge. To allow other components such as netback, the NIC driver, and network stack make progress, our application explicitly yields CPU time.

4.4 Blkback Driver

Initialization. First, blkback advertises its properties (the number of sectors, sector size, read/write mode, and features such as cache flush support, persistent grant references, the maximum number of indirect segments) via xenstore. Then, blkback sets its status in xenstore as 'connected.' Next, blkfront writes its properties to xenstore: ring references, an event channel number, support for persistent grant references, and an I/O protocol. Blkback maps the ring buffer and sets an event handler for notifications from blkfront. Unlike networking, storage I/O uses one ring buffer and one event channel.

Request. As discussed in Section 3.3, we have a thread for request notifications. The thread copies the request and stores segment information from the copied requests. With the segment information, I/O buffers are constructed in batches. The grant references are mapped to the pages inside the storage domain's address space and are used as buffer block data. Once buffers for all consecutive segments from one or more requests are constructed, the device driver interface is called through the upper layer to perform I/O.

Response. Blkback sends a response to blkfront when the bottom layer hears back from the device driver regarding previously submitted I/O. NetBSD's buffer structure makes it possible to set a callback function, which the device driver calls when it completes the submitted operations. In the callback, blkback unmaps grant references (unless they are persistent) and sends responses back to blkfront with the success/failure status, and then destroys memory buffers. Blkback uses its event channel to issue a virtual interrupt to blkfront.

Indirect Segment and Persistent References. Indirect segments can be crucial for NVMe SSDs but they require mapping indirection. First, we map grant references to pages. Then, we parse these pages, where each of them may contain up to 512 indirect segments. (Linux currently supports at most 32 indirect segments; we also limit the number to 32.)

To implement persistent referencing, we map each page separately so that we can reuse these pages even if they do not maintain any sequence in other requests. We store grant references and addresses of corresponding mapped pages in a lookup data structure, which makes it possible to avoid re-mapping.

Table 2. Hardware configuration.

	Server	Client
CPU	Xeon E5-2695 2.20GHz	Core i5-6600K 3.50GHz
Cores	24 (w/ HyperThreading)	4
L1/L2	32/256 KB per core	32/256 KB per core
L3	30720 KB	6114 KB
Memory	64 GB	16 GB
Network	Intel 82599ES 10-Gigabit	Intel 82599ES 10-Gigabit
Storage	Samsung 970 EVO Plus 500GB NVMe	N/A

5 Evaluation

We do a security evaluation, which includes CVE and ROP gadget analysis, and a performance analysis to identify performance overheads, if any, due to our implementation.

We only compare against Linux-based driver VMs since Linux is the only OS that supports driver domains. (NetBSD currently does not support them.) Likewise, Xen is the only hypervisor that has fully-fledged driver VM support currently. Our performance evaluation shows that, even with the mentioned challenges in Section 3.1, Kite is as performant as Linux for most of the cases. Some performance gains can be attributed to NetBSD drivers. Other gains come from the elimination of extra OS layers and user space.

Table 2 shows our setup. A client and server machines are directly connected by a SFI/SFP+ network cable. Driver VMs are tested on the server side. For network-related tests, our client acts as a load generator. Our server runs Xen (Dom0 has no storage/network drivers). Each server application runs in DomU. We create Linux-based and Kite driver VMs, which access the NIC and NVMe storage via PCI passthrough.

Dom0, DomU, and Linux-based driver domains run Ubuntu 18.04.3 LTS, kernel 5.0.0-23-generic. They are assigned 8GB, 5GB, and 2GB of RAM, respectively. For Kite VMs, we use Rumprun-SMP [43, 58] (based on NetBSD 9.0) with our additional changes. We assign less memory (1GB) since rumprun's footprint is smaller. In our tests, we found that one virtual CPU (vCPU) suffices since driver domains are I/O-intensive, but we support multiple vCPUs. Each driver domain allots 1 vCPU, and DomU allots 22 vCPUs. We use Ubuntu's standard Xen 4.9 hypervisor.

5.1 Security

5.1.1 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

Table 3. 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 local 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 a limited range of kernel memory.
2017-18344	timer_create	Allows userspace applications to read arbitrary kernel memory.
2017-17053	modify_ldt, clone	Allows an attacker to achieve a use-after-free impact 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 a parent process.
2009-0835	chmod, stat	Allows local users to bypass intended access restrictions via crafted syscalls.

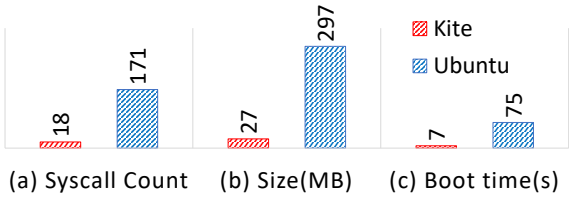


Figure 4. System call, size, and boot time comparison.

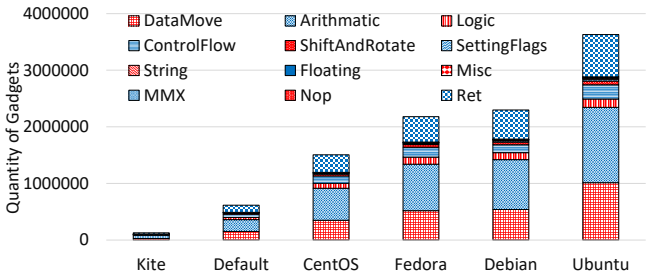


Figure 5. ROP gadget comparison.

and applications that are part of traditional service VMs. We found 172 [19] and 92 [20] 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.

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 4a). 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 3. Though we can block a few syscalls in Linux, lots of them are essential to initialize and run driver domains and cannot be removed.

5.1.2 ROP Gadget Reduction. Though reduction of ROP gadgets does not always improve security, a smaller number of ROP gadgets indicates potential obstacles for an attacker since the attacker will have a hard time constructing ROP chains to exploit a known vulnerability, assuming that NetBSD's code quality is generally on par with that of Linux. Using the methodology from Follner et al. [36] and a tool [12], we count ROP 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 5 breaks downs ROP gadgets, shown in Introduction, according to their category. The default Linux configuration is very minimal and has almost no modules, but already has 4x gadgets than Kite VMs. Note that for driver domains, we also need corresponding driver modules. Moreover, we do not count Linux user-space ROP gadgets here. This shows Kite's great potential for improved security.

5.2 Image Size and Boot Time

We already mentioned that rumprun is smaller than even a fairly minimal Linux image. To further elaborate, we compare image sizes of Kite and Linux-based driver domains used in experiments. For Kite, we measured the size of the entire Kite VM binary. For Linux, we measured only the size of the kernel and its modules, i.e., did not include the size of user-space programs. As Figure 4b shows, the Linux image is about 10x bigger than the Kite image.

Since boot times directly affect deployment in the cloud infrastructure, it is crucial to reduce them as much as possible. Moreover, driver domains can potentially be restarted when recovering from failures, where faster boot times are equally important. As Figure 4c shows, Kite takes 7 seconds to boot a driver domain. In contrast, Linux needs 75 seconds.

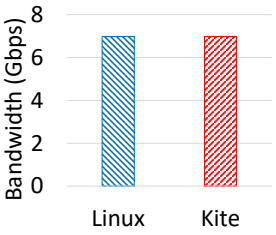


Figure 6. Nuttcp throughput for UDP file transfers.

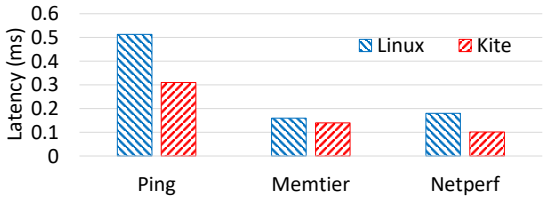


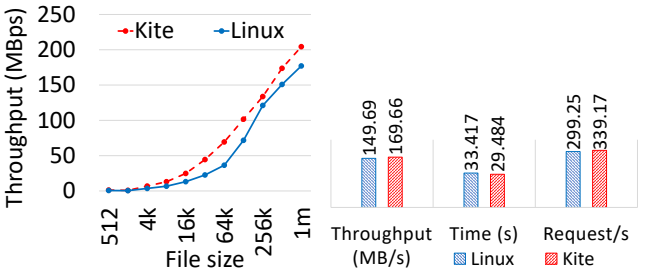
Figure 7. Linux vs. Kite network driver domains latency.

5.3 Network Domain

We have evaluated Kite that is only built for the network driver domain, discarding all unrelated components. 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. In this section, we evaluate our network domain, attached to a 10GbE NIC, using both micro- and macrobenchmarks. We run the nuttcp [9] microbenchmark to measure overall network throughput. Me measure network latency using ping, Netperf [7], and memtier [8] benchmark. We use macrobenchmarks including ApacheBench [72], Redis [11], and MySQL [6] to measure the performance of real-life applications, which can be relevant to cloud users.

5.3.1 Nuttcp. We measure the network throughput of a Linux guest machine when using 10GbE NIC through the Linux and Kite driver domain. To achieve optimal throughput with minimal packet loss, we run nuttcp benchmark [9] (v8.2.2) in the UDP mode with 4MB of window size and 8KB of buffer size. As shown in Figure 6, with the described configuration, we achieve about 7Gbps with less than 1.5% UDP packet loss for both Linux and Kite network domain.

5.3.2 Network Latency. We use various tools with different configurations for measuring network latency. Figure 7 shows the latency comparison when using Linux's and Kite's versions of netback. Pinging the guest machine from the client machine 100 times with one-second interval, we get lower latency for Kite (0.31ms) than for Linux (0.51ms). The Netperf [7] 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 [8], a benchmark for Memcached [5] reports 0.16ms and 0.15ms for Linux and Kite, respectively,



(a) Varying file size. **(b)** Throughput, transfer time, and request rate.

Figure 8. Apache server throughput.

Table 4. Relative standard deviation for experiments.

	Apache	Redis	Memtier	Sysbench
Linux	1.20%	0.00053%	0.0029%	0.0167%
Kite	1.44%	0.0011%	0.0023%	0.0496%

when performing 100000 SET and GET operations with a ratio of 1:10 and data of size 8KB.

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

5.3.3 Apache. To evaluate an HTTP server, we run Apache (v2.4.29) in DomU and Apache benchmark in the client machine. The server data (files) are randomly generated. The benchmark sends 100,000 requests and measures the server-side throughput. Each experiment is repeated 3 times. Figure 8a shows results for file sizes ranging from 512B to 1MB.

Figure 8b shows different parameters such as the transfer time, throughput, and request handling rate (logarithmic scale) for a specific Apache server experiment with a file of 512KB size. The Apache benchmark sends 100,000 requests with 40 concurrent requests. Kite is marginally faster. For the Linux and Kite results, the maximum relative standard deviation (RSD) is 1.20% and 1.44%, respectively.

5.3.4 Redis. We run Redis server [11], a well-known key-value store, in DomU to compare driver domains. We execute Redis benchmark (v4.0.9) with millions of SET/GET operations in the pipeline mode in the client machine. We set the pipeline size to 1,000. We also vary concurrency, wherein each GET/SET operation reads/writes 128MB of data with each key of size of 64 bits.

Figure 9 shows the number of SET/GET operations per second, on a logarithmic scale. Overall, Kite and Linux netback exhibit similar performance. The RSD for Linux's and Kite's netback is 0.00053% and 0.0011%, respectively.

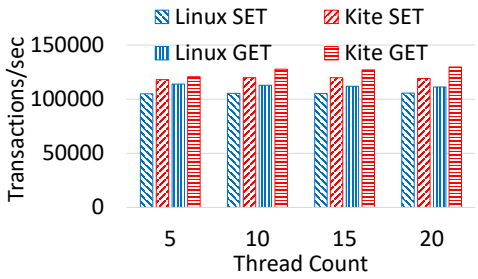
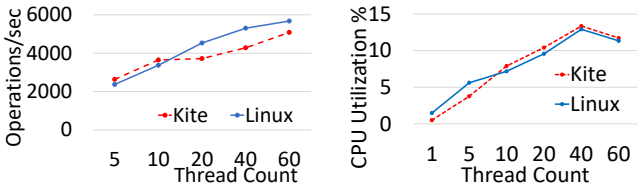


Figure 9. Redis key-value store throughput.



(a) Throughput varying threads. **(b)** CPU utilization.

Figure 10. MySQL throughput (network domain).

5.3.5 MySQL. Aside from NoSQL (Redis, Memcached), we evaluated the MySQL [6] server (v5.7.29), a popular SQL database, running on DomU. On the client machine, we ran Sysbench (v1.1.0) to measure the database 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). The benchmark sends read-only SQL queries to the server, which allows us to stress-test the network path.

Figure 10a shows the number of operations (queries and transactions). There is almost no performance difference when using Linux's or Kite's netback. The RSD is 0.0167% and 0.0496% for Linux and Kite, respectively. Figure 10b shows the average CPU utilization of DomU, measured using the sysstat utility [15], during the aforementioned benchmark execution. We found that DomU's CPU utilization for both Linux and Kite is very similar.

Table 4 shows the standard deviation of experiments described in Figures 7, 8, 9, and 10.

5.4 Storage Domain

For the storage domain performance evaluation, we have built Kite exclusively for storage domain. We use dd [3] as a microbenchmark. For macrobenchmarks, we use SysBench (v1.1.0) and Filebench [71] (v1.5-alpha3). Macrobenchmarks measure the performance of real-life applications such as MySQL and MongoDB database servers, fileserver, and webserver. For each run, we flush the read buffer and use total I/O size bigger than the main memory so that we exercise our storage domain more aggressively.

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.

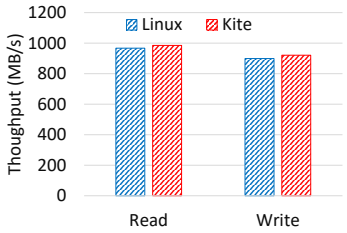
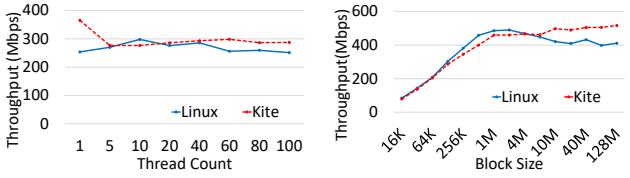


Figure 11. Storage throughput (dd).



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

Figure 12. SysBench file I/O throughput.

5.4.1 DD. To keep both reading and writing overheads minimal, we use `/dev/zero` as the source as well as the destination. We run experiments 3 times. Each time, 10GB of data is transferred from/to the device. Figure 11 shows similar performance for Linux and Kite.

5.4.2 SysBench File I/O. We use SysBench to measure file I/O performance. SysBench uses 192 files totaling 15GB. We perform 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 run takes 5 minutes.

Figure 12a shows throughputs for runs with a different number of threads for a block size of 256KB. Figure 12b shows throughputs for runs with a fixed number of threads (20) and different block sizes. Throughputs for Kite are very comparable to that of Linux. Kite is 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, respectively.

5.4.3 SysBench MySQL. We evaluate MySQL for storage using SysBench. Our database contains 100 tables, each with 1 million records, totalling 20GB of disk space. We vary the number of threads, each thread performs complex SQL queries [14]. Results in Figure 13 are identical.

5.4.4 Filebench Fileserver. To generate a file server workload, we use a fileserver benchmark from Filebench. We run 50 threads in parallel each performing series of operations including create, read, write, append, close, stat, delete, etc. Before running the workload, Filebench creates 100,000 files with an average size of 128KB, which makes a total of around 13GB. The mean append size is 1KB where the I/O sizes are varied, from 16KB to 8MB, for each run of 5 minutes.

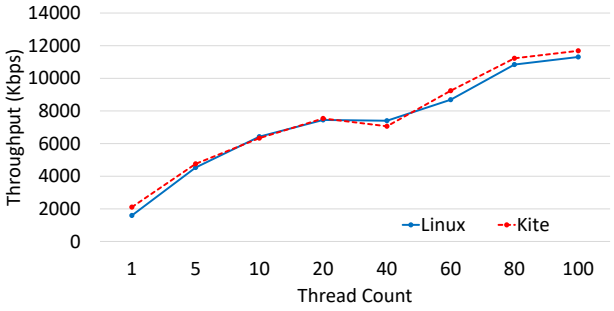


Figure 13. MySQL (storage domain).

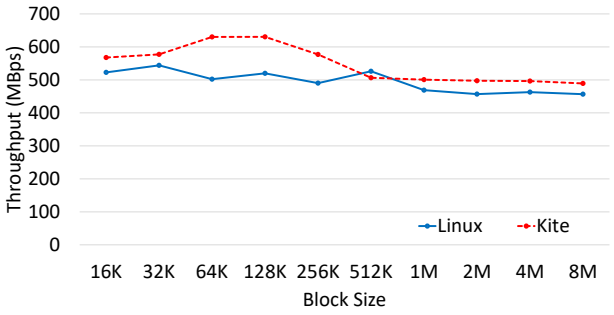


Figure 14. Filebench's fileserver throughput.

Throughput results are presented in Figure 14. Kite's storage domain often performs slightly better than Linux. The maximum incurred latency for this experiment is 8.99ms and 7.93ms for Linux and Kite, respectively.

5.4.5 Filebench MongoDB Server. We also evaluate MongoDB, a NoSQL database server, using Filebench because of its different file access patterns. We create 20GB of data with the mean I/O size of 4MB. Figure 15 shows the throughput, execution time per operation, and latency, stretched in 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.

5.4.6 Filebench Webserver. To generate a web server workload, we run 50 threads in parallel, where each thread performs a series of operations combining open, read, and close. First, filebench creates 200,000 files with an average size of 64KB, totaling around 13GB. The mean append size and I/O size is 16KB and 1MB, respectively. We run each experiment for 5 minutes. Figure 16 shows the web server throughput, execution time per operation, and latency, which are stretched in the logarithmic scale. Kite's storage domain takes a little less time than Linux for executing each operation, and thus Kite provides slightly higher throughput and lower latency.

5.5 Daemon Service VM

We measured our rumprun-based OpenDHCP server [60] performance using perfdhcp [10] (a DHCP benchmarking tool) and compared latencies with Linux. The average delay

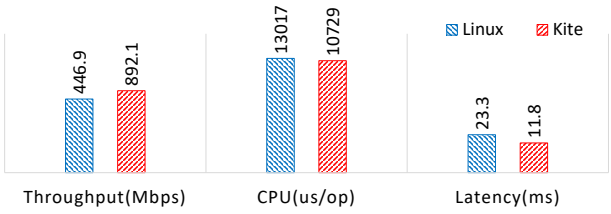


Figure 15. Filebench: MongoDB server.

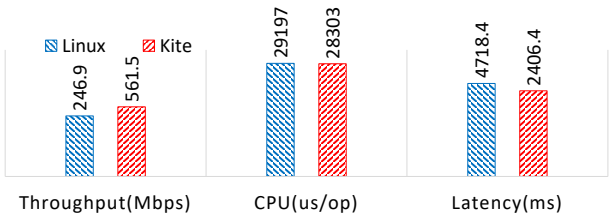


Figure 16. Filebench: Webserver.

for DHCP discover offer and request acknowledgement is very similar for rumprun and Linux (delays are ≈0.78 ms and ≈0.7 ms for Discover-Offer and Request-Ack, respectively).

6 Related Work

MLS Systems. Qubes OS [65] and SecureView [24] are Xen-based OSs for desktop and enterprise users, respectively. Qubes OS uses several types of VMs: network VM, storage VM, other service VMs, apps VMs, and administrative/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 runs blkback and provides access to the disk for the apps VMs. The administrative/GUI VM provides GUI to users. For all VMs, Qubes OS runs Linux. Kite's service VMs reduce memory footprints, startup times, and attack surface, which are critical for desktop and cloud users alike.

Some recent works [57] can strengthen the security of MLS OSs by continuously re-rerandomizing address space layout of device drivers. Kite can adopt similar techniques in the future.

Hypervisor Disaggregation. The Flux OSKit [37] demonstrates how OSs can be constructed using components from multiple different OSs, in particular by writing thin glue layers that can be used to leverage existent stable device drivers from Linux and BSD. However, the Flux OSKit requires manually modifying the glue layer each time drivers from source OSs change. In contrast, Kite is based on NetBSD's anykernel architecture, which factors out device drivers into components that can be executed unmodified anywhere, e.g., in other OSs, by using the rump kernel glue layer. Moreover, rumprun, the rump kernel-based unikernel, allows running a lightweight library OS atop the Xen hypervisor, unlike the Flux OSKit.

Hypervisors, which run multiple virtual machines on the same physical machine, are counted towards a trusted computing base (TCB) for cloud infrastructures. Xen 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 adversely affect any (DomU) guest OS. Therefore, there have been several efforts [31, 55, 68] for splitting Xen responsibilities, so that an exploited or failed component does not affect other components.

Prior works, discussed below, need *fully-fledged* VMs for drivers. Kite nicely complements them by targeting drivers.

Xoar [31] 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. Xoar uses nanOS only during bootstrapping, i.e., it does not contribute to further PCI communication. For network and storage driver VMs, Xoar uses heavyweight Linux, unlike our work. This still opens a potentially large attack surface. Our work nicely complements Xoar by providing lightweight driver domains.

The Nexen [68] architecture decomposes the hypervisor into three parts using page-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 DomU. Each slice serves only one DomU. The shared service domain provides the functionalities that could not be decomposed into slices. A 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. [55] 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 [30] 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, Dom0 has a relatively larger attack surface, and errors can affect core functionalities.

OS Architectures. Containers, e.g., Docker [54], have only process-level isolation. For better isolation, containers run in VMs with fully-fledged OSs [33, 38], but they have a large attack surface.

The hypervisor can also be used to build *one* OS by disaggregating its components. For example, VirtuOS [56] isolates critical OS components in separate VMs, which are similar to driver VMs. Since VirtuOS uses fully-fledged Linux-based VMs, the challenges with a large attack surface remain.

Unikernels are popular for cloud infrastructures [1, 63]. LibrettOS [58] extends rumprun's support for multicore systems and (partially) Xen HVM. Kite further extends LibrettOS's version of rumprun. Rumprun is ported to seL4 [34] but the same problem with driver domains (sharing the NIC between applications) arises in seL4. HEXO [59] takes advantage of low resource requirements of the HermitCore [48] unikernel. However, HermitCore and other unikernels lack rumprun's rich device driver support. None of the prior unikernel works improve the hypervisor itself. Kite's shows unikernels' performance, security, and resource consumption benefits for privileged hypervisor components (i.e., driver VMs).

Trusted Execution Environments (TEE), such as Intel SGX and AMD SEV, are being leveraged to secure applications running in general-purpose VMs [62, 69] and in unikernels [66, 74–76]. Several works [50, 77] proposed secure I/O paths between the applications and drivers as an extension for TEE. However, TEE is orthogonal to our work but can potentially be considered in Kite VMs to further enhance security.

Kernel-bypass libraries such as DPDK [73] and SPDK [70] provide high performance, but they lack standardized APIs (use custom APIs) and therefore incur prohibitive engineering efforts to modify existing applications to use them. As standardized API support for kernel-bypass libraries matures, SPDK or DPDK driver domains can be developed. This is an interesting future direction.

Our evaluation is inclusive of related works, such as Xoar and Nexen, since these disaggregation approaches use full-fledged Linux for driver domains. Therefore, we did not separately discuss them in the evaluation section.

7 Conclusions

In this paper, we presented the first implementation of unikernelized service VMs in MLS OSs. While past efforts have explored reducing the attack surface of hypervisors, our work is the first to focus on improving memory footprint, isolation, and security of privileged components such as device drivers that run outside of the hypervisor. Kite's novelty is that it does not need a full-blown OS to run driver domains. Though Kite's benefits partially come from the rumprun design itself, building a driver domain from rumprun was previously impossible. Kite's driver domains have a number of advantages compared to Linux-based driver domains: reduced number of syscalls, ROP gadgets, smaller image size, and faster boot time. Moreover, Kite does not rely on heavy-weight Linux tools (xen-tools), which present a number of security issues.

To realize the vision of unikernelized driver domains, we had to overcome many challenges such as extending rumprun's Xen HVM support, and implementing the netback and blkback drivers and special configuration/orchestration tools inside the rumprun unikernel. Our experimental evaluation reveals that our driver domains provide competitive performance to that of Linux-based driver domains, while retaining all the benefits of unikernels.

AVAILABILITY

Kite's code is available at https://github.com/ssrg-vt/kite.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Rodrigo Fonseca for their insightful comments and suggestions, which helped greatly improve this paper.

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 and N00014-19-1-2493.

Kite's DHCP server was integrated into an enterprise-level software infrastructure called SAVIOR (Secure Applications in Virtual Instantiations of Roles) system, which was developed as part of the IARPA VirtUE (Virtuous User Environment) program [17]. SAVIOR's source code repository is publicly available [13].

References

[1] 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.

[2] 2021. CVE-2013-2072. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2072.

[3] 2021. dd – convert and copy a file. https://man7.org/linux/man-pages/man1/dd.1.html.

[4] 2021. libxl CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libxl.

[5] 2021. Memcached. http://memcached.org/.

[6] 2021. MySQL. https://www.mysql.com/.

[7] 2021. Netperf Manual. http://www.cs.kent.edu/~farrell/dist/ref/Netperf.html.

[8] 2021. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark/.

[9] 2021. Nuttcp Welcome Page. https://www.nuttcp.net/Welcome%20Page.html.

[10] 2021. perfdhcp – DHCP benchmarking tool. http://manpages.ubuntu.com/manpages/xenial/man8/perfdhcp.8.html.

[11] 2021. Redis. https://redis.io/.

[12] 2021. Ropper. https://github.com/sashs/Ropper.

[13] 2021. SAVIOR (Secure Applications in Virtual Instantiations of Roles). https://github.com/NextCenturyCorporation/VirtUE.

[14] 2021. SysBench Manual. https://man7.org/linux/man-pages/man1/dd.1.html.

[15] 2021. SYSSTAT Utilities. http://sebastien.godard.pagesperso-orange.fr/.

[16] 2021. The NetBSD Project. https://netbsd.org.

[17] 2021. VirtUE (Virtuous User Environment). https://www.iarpa.gov/index.php/research-programs/virtue.

[18] 2021. Xen application CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=xen+application.

[19] 2021. Xen application CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+crafted+application.

[20] 2021. Xen application CVE search. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+shell.

[21] 2021. Xen block header. https://github.com/xen-project/xen/blob/master/xen/include/public/io/blkif.h.

[22] 2021. xen-utils-4.9. https://packages.ubuntu.com/bionic/xen-utils-4.9.

[23] 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).

[24] Air Force Research Laboratory AFRL/RIEB. 2021. SecureView. https://www.ainfosec.com/technologies/secureview/.

[25] AMD, Inc. 2021. AMD I/O Virtualization Technology (IOMMU) Specification. http://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf.

[26] 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

[27] Mark Beaumont, Jim McCarthy, and Toby Murray. 2016. The Cross Domain Desktop Compositor: Using Hardware-Based Video Compositing for a Multi-Level Secure User Interface. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. 533–545. https://doi.org/10.1145/2991079.2991087

[28] 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. https://doi.org/10.1109/CloudCom.2015.89

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

[30] 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* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2382196.2382226

[31] 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

[32] Raytheon Company. 2021. Raytheon Trusted Thin Client. https://www.raytheon.com/sites/default/files/capabilities/rtnwcm/groups/gallery/documents/digitalasset/rtn_216411.pdf.

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

[34] Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser. 2017. A Performance Evaluation of Rump Kernels As a Multi-server OS Building Block on seL4. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Article 11, 8 pages. https://doi.org/10.1145/3124680.3124727

[35] N. Feske and C. Helmuth. 2005. A Nitpicker's guide to a minimal-complexity secure GUI. In *21st Annual Computer Security Applications Conference (ACSAC '05)*. 85–94. https://doi.org/10.1109/CSAC.2005.7

[36] 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

[37] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 38–51. https://doi.org/10.1145/268998.266642

[38] Cloud Native Computing Foundation. 2021. Production-Grade Container Orchestration. https://kubernetes.io/.

[39] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and M. Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *RAID 2020*.

[40] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[41] Intel Corporation. 2021. 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.

[42] Abdullah Issa, Toby Murray, and Gidon Ernst. 2018. In Search of Perfect Users: Towards Understanding the Usability of Converged Multi-Level Secure User Interfaces. In *Proceedings of the 30th Australian Conference on Computer-Human Interaction* (Melbourne, Australia) *(OzCHI '18)*. 572–576. https://doi.org/10.1145/3292147.3292231

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

[44] 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.

[45] Avi Kivity. 2007. KVM: the Linux virtual machine monitor. In *2007 Ottawa Linux Symposium*. 225–230.

[46] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA) *(ATC '14)*. USENIX Association, USA, 61–72.

[47] 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

[48] 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* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2931088.2931093

[49] 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). https://doi.org/10.1109/IEEESTD.2019.8604150

[50] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. 2020. Establishing Trusted I/O Paths for SGX Client Systems With Aurora. *IEEE Transactions on Information Forensics and Security* 15 (2020), 1589–1600. https://doi.org/10.1109/TIFS.2019.2945621

[51] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon 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* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[52] 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

[53] 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

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

[55] 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* (Seattle, WA, USA) *(VEE '08)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/1346256.1346278

[56] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 116–132. https://doi.org/10.1145/2517349.2522719

[57] Ruslan Nikolaev, Hassan Nadeem, Cathlyn Stone, and Binoy Ravindran. 2022. Adelie: Continuous Address Space Layout Re-Randomization for Linux Drivers. 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, 483–498. https://doi.org/10.1145/3503222.3507779

[58] 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

[59] Pierre Olivier, A. K. M. 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* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/3307681.3325408

[60] OpenDHCP Server. 2021. http://dhcpserver.sourceforge.net/.

[61] 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

[62] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *ArXiv* abs/1908.11143 (2019).

[63] 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

[64] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

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

[66] Ioannis Sfyrakis and Thomas Gross. 2018. UniGuard: Protecting Unikernels Using Intel SGX. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 99–105. https://doi.org/10.1109/IC2E.2018.00032

[67] 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)*. 552–561. https://doi.org/10.1145/1315245.1315313

[68] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *NDSS '17*. https://doi.org/10.14722/ndss.2017.23455

[69] Shweta Shinde, Dat Le, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS '17*. https://doi.org/10.14722/ndss.2017.23500

[70] SPDK Contributors. 2021. Storage Performance Development Kit (SPDK). http://spdk.io/.

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

[72] The Apache Software Foundation. 2021. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[73] The Linux Foundation. 2021. Data Plane Development Kit (DPDK). http://dpdk.org/.

[74] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference* (Siena, Italy) *(CF'17)*. Association for Computing Machinery, New York, NY, USA, 35–44. https://doi.org/10.1145/3075564.3075572

[75] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the 9th European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

[76] 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

[77] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic Trusted I/O Path for Intel SGX *(CODASPY '17)*. Association for Computing Machinery, New York, NY, USA, 261–268. https://doi.org/10.1145/3029806.3029822

[78] Xen Project. 2014. Xenstored. https://wiki.xen.org/wiki/Xenstored.

[79] Xen Project. 2015. XenBus. https://wiki.xen.org/wiki/XenBus.

[80] Xen Project. 2015. XenStore. https://wiki.xen.org/wiki/XenStore.

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

[82] Xen Project. 2019. PCI Passthrough. https://wiki.xenproject.org/wiki/Xen_PCI_Passthrough.

[83] 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)*.

A Artifact Appendix

A.1 Abstract

In this Appendix, we discuss how to deploy Linux-based (Ubuntu) and Kite network and storage domains on a physical machine. We also discuss how to reproduce our experimental results presented in Section 5.

A.2 Description & Requirements

Section 5 describes the experimental setup, including the hardware, Xen hypervisor version, and operating systems used for the different domains. Here we describe other hardware and software dependencies, used benchmarks, and how to set up the artifact evaluation environment.

A.2.1 How to Access. The artifact is available at https://doi.org/10.5281/zenodo.6348173.

The artifact instructions are provided in README. The artifact evaluation files are located at the *kite/Artifact* directory, benchmarking scripts at *kite/Artifact/benchmarking_scripts*, and configuration files at *kite/Artifact/config*. The *config* directory contains configuration scripts for building the Ubuntu driver domain and guest domain. It also contains configuration scripts for booting up Ubuntu and Kite domains for network and storage domain evaluation in *network* and *storage* subdirectories, respectively.

Kite's latest source code is also available at https://github.com/ssrg-vt/kite.

A.2.2 Hardware Dependencies. Driver domains require physical 10Gbps NIC and NVMe devices via PCI passthrough (similar to Section 5). Moreover, virtualization support to run Xen is required. Physical machine deployment is required since deployment in a virtual machine would involve nested virtualization.

A.2.3 Software Dependencies. Kite should work with any Linux-based OS. However, we recommend Ubuntu 18.04 LTS for the Dom0 and DomU OSs.

A.2.4 Benchmarks. Our evaluation requires installation of Nuttcp, Netperf, Redis, Apache, Memcached, MySQL on the server machine inside DomU. The client (load generator) machine should have corresponding client benchmark applications for Nuttcp (v8.2.2), Netperf (v2.6.0-2.1), Redis (v4.0.9), Apache (v2.4.29), sysbench (v1.1.0) (for MySQL (v5.7.29)) for network domain evaluation. For storage domain evaluation, the client machine needs MySQL server (v5.7.29), sysbench benchmark (v1.1.0), and Filebench (1.5-alpha3) benchmarks. The benchmark scripts and instructions can be found in the artifact package.

A.3 Setup

A.3.1 Xen. First, install Ubuntu 18.04 LTS on a 64-bit x86 machine. Please select "Use LVM with the new Ubuntu installation."

Then, install the Xen 4.9.1 hypervisor and reboot the machine; GRUB should automatically boot Xen and launch Dom0:

```
# apt install xen-hypervisor-amd64
```

A.3.2 PCI Passthrough. Find BDF numbers of the available PCI devices (NIC, NVMe) using the `lspci` command. Then, add the corresponding device to the PCI assignable list, where `xx:xx.x` represents the BDF number:

```
# modprobe xen-pciback
# xl pci-assignable-add xx:xx.x
```

A.3.3 Kite. Please set up Kite's build environment:

```
# apt install build-essential git
# apt install libz-dev libxen-dev
```

Next, get Kite's source and build it:

```
# git clone https://github.com/ssrg-vt/kite
# cd kite
# git submodule update --init --recursive --remote
# CC=‘echo $PWD‘/gcc8fix.sh ./build-rr.sh -j16 hw
# cd bridge
# ./ifconf.sh && ./run.sh
# cd ../vbdconf
# ./run.sh
```

A.3.4 Guest Domain (DomU) for Server Applications. First, create a logical disk drive to install a guest OS:

```
# lvcreate -L 40G -n ubuntu_guest /dev/<VG>
```

Please download Ubuntu 18.04 LTS from https://releases.ubuntu.com/18.04/ubuntu-18.04.6-desktop-amd64.iso. Then launch a guest VM using the provided configuration file from the artifact package:

```
# xl create -c config/ubuntu_guest_setup.cfg
```

Install a VNC client (such as vncviewer) for GUI access (using `localhost`) and finish Ubuntu guest installation.

A.3.5 Linux Driver Domain. First, create a logical disk drive to install the Ubuntu driver domain:

```
# lvcreate -L 40G -n ubuntu_dd /dev/<VG>
```

Then copy the contents from `/dev/<VG>/ubuntu_guest` to save the OS installation time for the Ubuntu driver domain:

```
# dd if=/dev/<VG>/ubuntu_guest
 of=/dev/<VG>/ubuntu_dd bs=1G count=40
```

Launch the Ubuntu driver domain using the provided configuration file `ubuntu_dd_setup.cfg`:

```
# xl create -c config/ubuntu_dd_setup.cfg
```

Next, install Xen tools. (It can be easier to simply install the Xen hypervisor again.) Then, replace `/etc/default/grub.d/xen.cfg` with the provided `config/xen.cfg`; it will prevent the driver domain itself from booting the Xen hypervisor. Finally, update GRUB by running 'update-grub'.

A.4 Evaluation Workflow

A.4.1 Major Claims.

- (C1): Kite achieves 10x faster boot time than an Ubuntu-based driver domain. See experiment E1 in Section 5.2, for which results are reported in Figure 4c.
- *(*C2): Kite's network domain performs similarly to an Ubuntu-based network domain. See experiment E2.
- *(*C3): Kite's storage domain performs similarly to an Ubuntu-based storage domain. See experiment E3.
- *(*C4): We skip Figure 5 (ROP gadgets) due to the need of extra tools; this is not a fundamental paper result, it is just given for information purposes only. The reduced attack surface also follows from reduced image sizes.

A.4.2 Experiment E1 [Boot Time]. We measure the boot time for both Ubuntu and Kite driver domains. We use network domains but results are similar for storage domains.

First, update `config/network/ubunt_dd.cfg` with the BDF number of the network device:

```
pci=['xx:xx.x,permissive=1']
```

Then, launch an Ubuntu-based network domain and measure the boot time manually until you see the login screen:

```
# xl create -c config/network/ubuntu_dd.cfg
```

Next, terminate the Ubuntu domain. Then, to run Kite's network domain, first add the network device's BDF number into the `config/network/kite_dd.cfg` file. Next, launch the Kite network domain using the following command:

```
# xl create -c config/network/kite_dd.cfg
```

Measure the boot time manually until you see a notification that says `'Network domain is ready.'` To destroy Kite's domains, run the following:

```
# xl destroy <Kite domain id>
```

To locate domain IDs, run the following command in Dom0, where Kite's network domain is named 'netbackend':

```
# xl list
```

Kite should exhibit at least 10x faster boot time.

A.4.3 Experiment E2 [Network Performance]. To evaluate an Ubuntu-based network domain, first launch it:

```
# xl create -c config/network/ubuntu_dd.cfg
```

In the driver domain, create a network bridge, named `xenbr0`, with the network interface corresponding to the network device (assigned via PCI passthrough). Then, launch the Xen driver domain daemon:

```
# xl devd
```

Next, launch the Ubuntu DomU guest:

```
# xl create -c config/network/guest_on_ubuntu.cfg
```

We run server applications such as Apache, Redis, Memcached, and MySQL in this guest machine. The client machine should be connected to the same network. We use the benchmark scripts from the artifact package to measure network throughput, CPU utilization, and latency. (The details on the client machine setup are in README_benchmark.pdf.)

To evaluate Kite's network domain, launch Kite as explained in E1. Next, launch the Ubuntu DomU:

```
# xl create -c config/network/guest_on_kite.cfg
```

You can run the same network benchmark experiments from the client machine to evaluate Kite's network domain. We expect Kite to yield similar performance to that of Ubuntu.

A.4.4 Experiment E3 [Storage Performance]. To evaluate the Ubuntu storage domain, first launch it:

```
# xl create -c config/storage/ubuntu_dd.cfg
```

Attach the storage device to the Ubuntu storage domain:

```
# xl pci-attach <Driver Domain ID> 'xx:x.x'
```

Launch the Xen driver domain daemon:

```
# xl devd
```

Next, launch the Ubuntu DomU guest:

```
# xl create -c config/storage/guest_on_ubuntu.cfg
```

Mount the PV storage device to an empty directory:

```
# mkdir disk
# mount /dev/xvdb disk
```

We run the MySQL server with sysbench and Filebench benchmark for file server, web server, and the MongoDB server to evaluate the storage domains. You can use the benchmark scripts from the artifact package (instructions are in README_benchmark.pdf) to measure storage throughput, CPU utilization, and latency.

To evaluate Kite's storage domain, first change the storage device's BDF number in the `config/storage/kite_dd.cfg` file. Then, run the following command to build the storage domain application and launch Kite's storage domain:

```
# xl create -c config/storage/kite_dd.cfg
```

Next, launch the Ubuntu DomU guest:

```
# xl create -c config/storage/guest_on_kite.cfg
```

You can run the same benchmark experiments in the Ubuntu guest VM to evaluate Kite's storage domain. We expect Kite to yield performance similar to that of Ubuntu.

A.5 Notes on Reusability

Our paper requires a physical machine with 10Gbps NIC and NVMe (virtual machines, containers, etc. are impossible).