Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor

by

Mihir Sudarshan Nanavati

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

June 2011

Abstract

Virtualization platforms have grown with an increasing demand for new technologies, with the modern enterprise-ready virtualization platform being a complex, feature-rich piece of software. Despite the small size of hypervisors, the trusted computing base (TCB) of most enterprise platforms is larger than that of most monolithic commodity operating systems. Several key components of the Xen platform reside in a special, highly-privileged virtual machine or the “Control VM”. We present *Xoar*, a modified version of the Xen platform that retrofits the modularity and isolation principles championed by microkernels onto a mature virtualization platform.

Xoar divides the large, shared control VM of Xen’s TCB into a set of independent, isolated, single purpose components called *shards*. Shards improve security in several ways: components are restricted to the least privilege necessary for functioning and any sharing between guest VMs is explicitly configurable and auditable in tune with the desired risk exposure policies. Microrebooting components at configurable frequencies reduces the temporal attack surface.

Our approach does not require any existing functionality to be sacrificed and allows components to be reused rather than rewritten from scratch. The low performance overhead leads us to believe that Xoar is viable alternative for deployment in enterprise environments.

Preface

The work described in this thesis is part of a larger system, built collaboratively with Patrick Colp, Jun Zhu, Tim Deegan, George Coker, William Aiello, and Andrew Warfield, and has been submitted as a paper titled ”Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor” which is currently under review.

This project has been carried out under the supervision of Tim Deegan, Principal Software Engineer of the Xen Platform Team at Citrix Systems in Cambridge, and William Aiello and Andrew Warfield at UBC. nanOS, a minimal operating system, used by some of the components described in Chapter 5 is the work of George Coker and his team at the National Security Agency (NSA). The implementation of BlkBack, one of the driver domains in Section 5.4, is a collaboration with Jun Zhu, a fellow summer intern working under Tim Deegan at Citrix Systems. The patches required for the Linux kernel belong to him.

Section 5.2 describes the overall bootup process of Xoar. The interactions between the different components required to orchestrate boot was collectively designed and implemented with Patrick Colp at UBC. The deprivileging of XenStore using Grant Tables, one of the mechanisms described in Section 5.6, is entirely the work of Patrick Colp.

Section 3.3 is a brief overview of the snapshot and rollback mechanism developed by Patrick Colp and has been published separately as Patrick Clop. Eliminating the Long-Running Process: Separating Code and State. M.Sc. Thesis, 2010.

All other parts of the implementation, unless specified otherwise, are the work of the author.

Table of Contents

List of Tables

List of Figures

Acknowledgments

There are several people to whom I owe a debt of gratitude: My advisors, Bill Aiello and Andy Warfield, both for guiding my research and for handling frustrated ramblings with encouragement and humour. Norm Hutchinson for being warm and approachable at all times, in addition to being the second reader for this thesis.

The work described here is part of a larger system and would not have been possible without my co-authors – Tim Deegan, Patrick Colp, Jun Zhu, and George Coker. Tim, in particular, has patiently answered several questions about the intricacies of the Xen hypervisor.

Several friends and colleagues, too many to name individually, have been springboards for ideas and discussions and have provided good company and cheer through several long hours of coding. Finally, my family who have lovingly supplied the required combination of nagging and cheerleading in any endeavour of mine.

To my parents,
with love.

Chapter 1

Introduction

Hardware virtualization has spawned an industry of highly-utilised, low-cost, "cloud based" hosting solutions allowing attackers to co-locate malicious virtual machines on the same physical machine as production machines of other customers. The intuitive risks due to the heightened exposure in such environments are often overlooked in favour of the benefits of better utilisation of over-provisioned server hardware.

Proponents of virtualization have touted the small trusted computing base (TCB) of the hypervisor, along with the narrow, auditable interfaces exposed to the guest VMs as strong arguments in favour of the security of virtualized environments. Hypervisors have grown out of a need to support the features demanded by industry, the end result being that the TCBs of mature, deployed virtualization platforms as used in conventional cloud hosting environments are *larger* than that of commodity server operating systems. Components such as device drivers for physical hardware, device emulation, and the administrative toolstack are the often ignored shared services belonging to the TCB of the system. Microkernel based hypervisors like Xen [5] and Hyper-V [24], rely on a privileged virtual machine or control VM, usually running a full-fledged server OS, to host these components. This large, shared control VM is the "elephant in the room", and conveniently overlooked while discussing the security of these systems.

While a large TCB is not necessarily a source of worry in itself, the presence of the shared control VM represents a serious source of vulnerabilities for virtualization platforms. Xen's control VM hosts varied services like device drivers for hardware interaction, virtualized and emulated device providers, an administrative toolstack, *etc.* Each of these services exposes a non-standard, service-specific set of interfaces to the underlying hardware or other guest VMs. As all these components are co-located in the same monolithic TCB, a compromise of any of the components threatens the entire platform, with the security of the system being determined by the security of the weakest component.

Historically, operating systems have solved the problem of large, monolithic TCBs by breaking them up into small, isolated pieces, each of which has the least privilege necessary for a specific task [48]. Research projects using similar techniques to re-architect the platform have resulted in from-scratch hypervisors [25, 46] with improved security and minimal sharing. Unfortunately, this heightened security has almost universally led to sacrifices in functionality such as limited device driver support or limiting the density of VM hosting. This illustrates the fundamental tension between security and functionality in the design of commercial virtualization platforms. While the security of these platforms is paramount, it cannot compromise the rich set of features necessary for deployment in commercial hosting environments. The system described in this thesis avoids this compromise by partitioning the control VM of the monolithic TCB *without* sacrificing the efficiency or functionality of existing systems.

Efficient Resource Utilisation and Improved Management: Wide scale deployments of virtualization platforms in large-scale hosting facilities are largely motivated by the efficient virtualization of physical computing resources and the improved facilities for the administration and management of hosted virtual machines. Modern virtualization platforms are necessarily large and complex pieces of software because of a need to support these functional requirements, which must be accounted for while architecting any security improvements to the platform.

Datacenters have shifted to evaluating the efficiency of computing in terms of performance per Watt [3], rather than more traditional metrics like peak capacity. Cloud hosting providers, in particular, are inclined to reduce the costs of meeting their customer SLAs as far as possible, by aggressively multiplexing and oversubscribing their resources.

Despite the increasing availability of cores on a single physical processing unit [22], multiplexing individual CPUs leads to a considerable gain in efficiency. Best practices in virtual desktop deployments involve deploying 10 VMs per CPU core [50]. Further packing density is achieved by sharing identical pages of memory [21, 38, 51] and common blocks on disk [13, 20, 37] between VMs. These features require the virtualization platform to interpose on memory and device requests in order to transform and redirect requests.

Virtualization eases administrative load by decoupling the software machines from the underlying hardware resources. By presenting a unified abstraction of hardware to the operating system, it allows VMs to seamlessly be moved from one physical machine to another [12]. Live migration allows for hardware maintenance and replacement without affecting existing VMs, and is used to provide high availability in the face of unexpected failures [8, 16].

The Shard Abstraction: We introduce the abstraction of *shards* and partition the control VM of Xen into eight classes, potentially having multiple instances, of shards while maintaining both functional parity and legacy support with the current platform. Shards are isolated, self-contained virtual machines hosting components of the control VM. The interfaces and relationships between shards are explicit, allowing the administrator to apply different sharing policies to each component. Individual shards can be audited to better reason about the risks of exposure in case of compromises. Finally, shards limit the temporal attack surface of components by a judicious use of microreboots [10].

Chapter 2

TCBs, Trust and Threats

This work aims to strengthen the security of virtualization platforms in deployed, multi-tenant environments by reducing the size and complexity of the *Trusted Computing Base* (TCB) of the system. Our approach involves incorporating stronger isolation boundaries between existing components in the trusted computing base, increasing our ability to manage and reason about exposure to risk. This chapter describes the TCB of a typical enterprise virtualization platform and articulates our threat model, along with a classification of relevant existing published vulnerabilities in these environments. It concludes by describing existing approaches to increasing the security of the TCB.

2.1 TCBs: Trust and Exposure

The TCB is classically defined as "the totality of protection mechanisms within a computer system — including hardware, firmware, and software — the combination of which is responsible for enforcing a security policy" [1]. This rather vague definition results in the common misconception that there exists a single TCB for the entire system.

We believe that trust is *contextual*: different subsystems rely on and trust, to varying degrees, a different set of components in order to function correctly. A compromise of certain elements in a system's TCB may impact the confidentiality of all data on the system or only result in the loss of availability of a particular

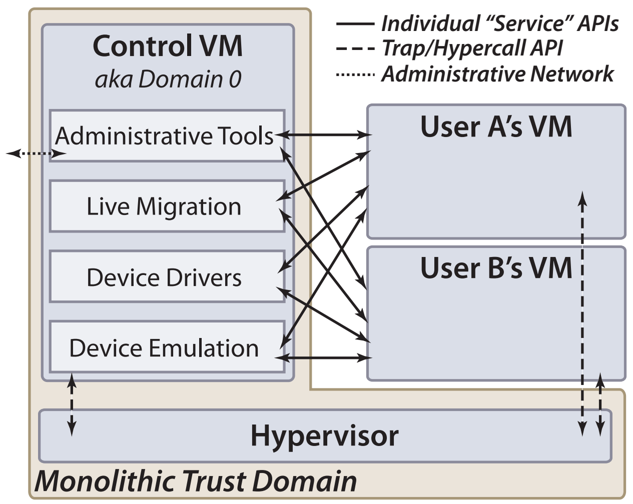

Figure 2.1: TCB of the Virtualization Platform

subsystem.

In line with existing work on TCB reduction, we define the TCB of a subsystem *S* as "the set of components that *S* trusts not to violate the security of *S*" [23, 39]. A compromise of any component in the TCB affords the attacker two benefits. First, they gain the privileges of that component, such as access to arbitrary regions of memory or control of hardware. Second, they gain access to other elements of the TCB which allows them to attempt to inject malicious requests or responses over those interfaces.

2.1.1 Virtualization Platforms and TCBs: A Guest's Perspective

Enterprise virtualization platforms, such as Xen, Hyper-V, and VMware ESX, are responsible for the isolation, scheduling and memory management of all hosted guest virtual machines. Hypervisors run in a more privileged hardware protection ring than hosted OSes, allowing them arbitrary access to the guest VMs' memory. The higher privileges of the hypervisor compared to the guests means that, along with the hardware, it forms a part of the TCB of the system.

Architecturally, these platforms are responsible for more than just CPU and memory virtualization. Device drivers and device emulation components manage and virtualize physical hardware for guest VMs. Management toolstacks are required to create and destroy VMs, or to dynamically reconfigure ones running on the system. Further components provide virtual consoles, configuration state management, memory compression and sharing, live migration, and other enterprise-level features. Commodity virtualization platforms, like the ones mentioned above, provide all these services from a single monolithic trust domain; either directly as part of the hypervisor or within a single privileged virtual machine, or control VM, running on it. Figure 2.1 illustrates an example of this monolithic trust domain as it is implemented in Xen.

2.2 Threat Model

We assume a well-managed and professionally administered virtualization platform that restricts access to both physical resources and the privileged administrative interfaces. We are specifically concerned with threats originating from within guest VMs hosted on the system. In a multi-tenant environment, guest VMs are owned by untrusted third parties and often exposed to the public Internet: we assume that they are both arbitrarily vulnerable and actively hostile to the underlying platform.

An attacker with a foothold within a guest VM may then attempt to compromise any interface presented by the virtualization platform. With the services exposing these interfaces often co-located, and the interfaces themselves shared between several mutually untrusting guests, a vulnerability in *any* of the interfaces could compromise the entire system. For instance, an attacker that compromises the virtual frame buffer can leverage that position to access video memory of other VMs that are being served from the same frame buffer provider [30].

Further, we assume that the underlying virtualization platform *will contain bugs* that are a potential source of compromise. Rather than exploring techniques that might allow for the construction of a bug-free platform, our more pragmatic goal is to provide an architecture that isolates components, thus containing compromises and limiting the capabilities of an attacker.

2.2.1 Example Attack Vectors

Most Xen distributions use the Carnegie Mellon Software Engineering Institute Computer Emergency Response Team's (CERT) vulnerability registry to publish known compromises. While some VMware-related incidents have been reported to CERT, VMware also maintains their own list of security advisories. In analyzing both of these registries for entries related to known virtualization software using Type-1 hypervisors,[1] we identified a total of 44 reported vulnerabilities.

Of the reported Xen vulnerabilities, 23 originated from within guest VMs, 12 of which were buffer overflows allowing arbitrary code execution with elevated privileges, while the other 11 were denial-of-service attacks. Classifying by attack vector showed 14 vulnerabilities in the device emulation layer, with another 4 in the virtualized device layer. The remainder included 4 in management components and a single hypervisor exploit, ironically in the security extensions. 22 of the 23 attacks outlined above are against service components in the control VM.

I/O virtualization may be realized through the use of device emulation, which allows the OS to use unmodified device drivers, or via virtualization-aware "paravirtual" drivers. Due to the inherent difficulty of exposing such device-agnostic abstractions to guests, both of these approaches result in complex, difficult to secure interfaces. Device emulation, in particular, is the target of several of these attacks. Unfortunately, the security of the entire system is only as good as that of the weakest component of the monolithic trust domain.

2.3 Existing Approaches to Securing the TCB

The widespread use of virtual machines has led to extensive studies on the security of virtualization platforms and several attempts to address the problem of securing the TCB have been made. We evaluate a number of these techniques in light of our functional requirements.

[1]There were a very large number of reports relating to Type-2 hypervisors, most of which assume the attacker has access to the host OS and compromises known host OS vulnerabilities — for instance, using Windows exploits to compromise VMware workstation. As these attacks are not representative of our threat model, we chose to exclude them from our analysis.

2.3.1 Build a Smaller Hypervisor

SecVisor [44] and BitVisor [45] are examples of tiny hypervisors, built with TCB size as a primary concern, that use the interposition capabilities of hypervisors to retrofit security features for commodity OSes. While significantly reducing the TCB of the system, these systems do not share the multi-tenancy goals of commodity hypervisors and are unsuitable for enterprise environments.

Nova [46] uses a microkernel-like architecture, explicitly partitioning the TCB into several user-level processes within the hypervisor. Although capable of running multiple VMs concurrently, it is far from complete: it cannot run Windows guests, and has limited hardware support because it does not reuse existing OS drivers.

NoHype [25] advocates removing the hypervisor altogether, using static partitioning of CPUs, memory, and peripherals among VMs. This would allow a host to be shared by multiple operating systems, but with none of the other benefits of virtualization. In particular, the virtualization layer could no longer be used for interposition, which is necessary for live migration, memory sharing and compression, and security enhancements.

2.3.2 Harden the Components of the TCB

The security of individual components of the TCB can be improved using a combination of improved code quality and access control checks to restrict the privilege of these components. Xen's XAPI toolstack is written in OCaml and benefits from the robustness that a statically typed, functional language provides [43]. Xen and Linux both have mechanisms to enforce fine-grained security policies [36, 42]. While useful, these techniques do not address the underlying concern about the size of the TCB.

2.3.3 Partition the TCB and Reduce Privilege of the Parts

Murray, *et al.* [39] removed Dom0 userspace from the TCB altogether by moving the VM builder into a separate privileged VM. While a step in the right direction, it does not provide functional parity with Xen, nor does it remove the Dom0 kernel from the TCB, leaving the system vulnerable to attacks on exposed interfaces like network drivers.

Driver domains [17] allow device drivers to be hosted in dedicated VMs rather than Dom0, resulting in better driver isolation. Qubes [41] uses driver domains in a single-user environment, but does not otherwise break up Dom0. Stub domains [49] help isolate the QEMU device model for improved performance and isolation. Xoar builds on this body of work and extends the ideas to cover the whole of the control VM.

Chapter 3

Design

Xoar aims to harden a well-deployed, feature-rich enterprise virtualization platform by strengthening the isolation boundaries between potentially vulnerable components of the monolithic TCB. A primary design goal is to maintain functional parity with the original system and complete legacy support with existing management and VM interfaces.

Partitioning is an intuitively attractive approach to improving the security of systems, but has not been widely used due to inherent performance overheads [2] and the increased complexity caused by the inter-component communication interfaces. In Xoar we observe that virtual machines in general, and, in particular, the shared components that run as services above the hypervisor are excellent candidates for isolation.

We achieve this isolation by introducing a specialised virtual machine based container: the *shard*. Shards are units of isolation hosting the service components of the control VM. They are regular guest VMs differing only in two key aspects; from the perspective of the hypervisor shards are the only virtual machines capable of invoking privileged functionality and accessing inter-VM communication channels. Shards provide services to other guest VMs and, aside from the hypervisor, are the only potentially shared components in the system.

Shards are entire virtual machines, capable of hosting full OSes and entire application stacks. Control VM based services like device drivers and ring-3 applications can be hosted in individual shards and removed from the monolithic TCB entirely. These shards are assigned the least privilege required by the service.

Components with heightened privilege and those with a high degree of sharing amongst guest VMs are attractive targets identified by the threat model laid out in the preceding chapter. Xoar confines both these high-risk qualities to a single abstraction, and then provides specific techniques to enhance the security of that particular abstraction.

Shards have been designed with the goal of both increasing the security of the virtualization platform and allowing users to better manage their exposure with an understanding of the risks of resource sharing. The use of shards results in three tangible benefits:

1. **Least-Privileged Isolation** Each component formed by partitioning the control VM of the system has the least privileges necessary to function correctly. Interfaces exposed by a component, both to dependent VMs and to the rest of the system, are limited to the minimal necessary set. This confines a successful attack to the limited capabilities and interfaces of the exploited component.

2. **Managed Exposure to Risk** Despite the desire to avoid sharing components across multiple VMs, some degree of sharing is often inevitable. Explicitly specifying this sharing allows customers to formulate risk exposure policies with a better understanding of the components being shared, and administrators to securely log the relationships between shared components. The audit log can be queried to identify potentially vulnerable machines after a compromise has been detected.

3. **Reduced Temporal Attack Surface** Shared components are restored to a known good state as frequently as is possible from a performance perspective by the use of microreboots [10]. Attackers that manage to exploit these components have limited execution time till the next reboot cycle.

The remainder of this chapter discusses the selection of the appropriate granularity and boundaries of isolation in creating shards. We then discuss two possible deployment scenarios for such a system: A densely-multiplexed public cloud

assign_pci_device (*PCI_domain*, *bus*, *slot*)
permit_hypercall (*hypercall_id*)
allow_delegation (*guest_id*)

Figure 3.1: Privilege Assignment API

configuration similar to currently deployed large-scale commercial hosting environments, and a coarse-grained resource partitioned configuration, appropriate for private clouds in which separate administrators receive physical resources that can be independently managed.

3.1 Isolation: Shards of a Broken TCB

Choosing the appropriate shards for a particular control VM is an inexact science, guided by the desire to achieve better isolation of privilege, sharing and execution state. Our design is based on our understanding of Xen's TCB, but the general principles, enumerated below, should be applicable to any other Type-1 virtualization platform.

1. **Minimal Privilege** Each component should obey the principle of least privilege. A component requiring heightened privileges for a subset of its functionality should be divided into a privileged and non-privileged shard.

2. **Minimal Sharing** Components should be independently restartable, avoiding tight coupling and heavy inter-component communications. Tightly coupled components should be co-located in the same shard, unless requiring different capabilities.

3. **Reflect Hardware Abstractions** The virtualization platform exposes the familiar abstraction of physical hardware devices to OSes using virtualized or emulated devices. Self-contained, independent hardware resources designed to be shared amongst users are an excellent fit for the shard abstraction.

A VM is configured to be a shard using a `shard` block in its config file. This block indicates that the VM can be assigned additional privileges and contains parameters that describe these capabilities.

Figure 3.1 shows an API for the assignment of the three configurable privileges: direct hardware assignment, privileged hypercalls, and the administrative privileges delegated to another guest VM.

Several virtualization platforms allow guest VMs to access hardware devices directly. Given a PCI domain, a bus, and a slot number, the hypervisor checks the availability of the device to ensure it is not already assigned to another VM. Once validated, the device is passed through to the guest VM, which interfaces directly with the hardware.

Hypercall permissions specify if a shard requires heightened privileges and access to some of the privileged hypercalls provided by the hypervisor. By whitelisting only the necessary privileged hypercalls to allow in addition to the default unprivileged ones available to all guest VMs, we ensure that each shard has the least required privileges.

A shard may delegate its administrative privileges to another guest, granting that guest VM the ability to administer the shard. This is illustrated in the private cloud scenario described in Section 3.4, in which each user of a system is assigned their own administrative toolstack and is able to manage both their own hosted VMs and the shards that support them.

3.2 Sharing: Manage and Reason about Exposure

Shards allow an explicit description of the interdependencies between shared components of the control VM, in addition to confining and limiting the attack surface of these services. Guests can restrict exposure only to trusted shards by the use of configuration constraints, or reason about severity and consequences of compromises after they occur.

3.2.1 Configuration Constraints

Guest VMs can constrain the use of shards to be limited to a certain groups of VMs, preventing sharing with untrusted machines. The `constrain_group` configuration parameter specifies that all shared shards *must* only be shared amongst VMs with similar configuration constraints. Xoar enforces this policy, ensuring that no two VMs with differing constraints share the same shard. In case there is a lack of appropriate shards, VM creation fails rather than forcing the guest VM into an undesired sharing configuration.

This allows users to specify constraint tags on all their hosted VMs and restrict the sharing of shards to within guests under their control, effectively limiting their exposure to third-party virtual machines.

3.2.2 Secure Audit

Auditing and system forensic techniques that help reconstruct an attack often generate dependency graphs to depict the relationships between objects over time [27]. The component level isolation and explicit description of the relationships between shards simplify the generation of these dependency graphs. Xoar uses techniques similar to Taser [19] for system forensics. Events such as the creation, destruction and migration of VMs, along with all the shards linked to the VM are stored in an off-host, append-only audit log.

Two simple examples help illustrate the uses of such audit logs. In case of compromised shards, enumerating all guest VMs that relied on that particular service at any point of time during the compromise can help to identify and notify potentially affected customers.

OS kernels and device drivers are often patched and updated, sometimes as frequently as every week. Shards allow services to be upgraded independently at differing intervals. In the event of a vulnerability being discovered in a specific release of a component after the fact, the audit log may be used to identify all guest VMs that were serviced by a vulnerable shard.

3.3 Freshness: Protect VMs in Time

Shards help defend the *temporal* attack surface of components, periodically discarding existing execution state through the use of frequent restarts. Several control VM based services are necessarily long-lived; once compromised, attackers have essentially unlimited time to attack the infrastructure or other guest VMs. Frequently resetting services to a known good state forces attackers to constantly re-compromise these components, and temporally limits the exposure to the end of the current execution cycle. Existing work on microreboots and "crash-only soft-

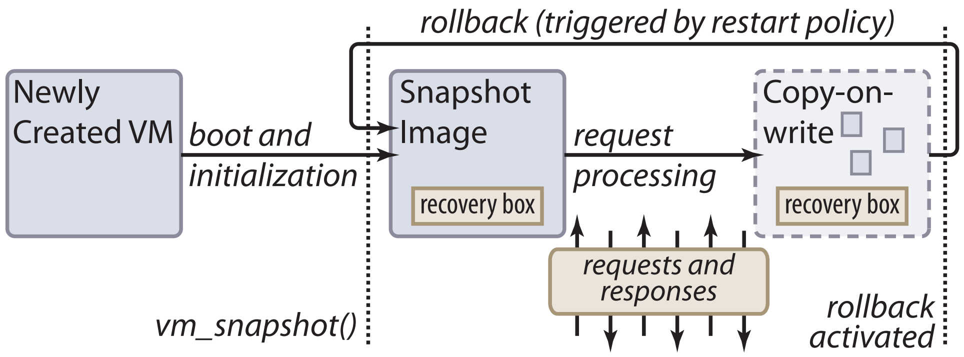

Figure 3.2: Snapshot and Rollback Mechanism

ware" have also argued that it is easier to reason about the correctness of a program at the start of day, prior to the inevitable state space explosion accompanying long periods of execution.

Virtual machine protocols frequently deal with disconnection and renegotiation of connections during live migration and are designed to cache and retransmit failed requests, making virtual machines the ideal container to reboot. There are, unfortunately, two major challenges associated with microreboots:

1. Full reboots are slow and significantly reduce performance, especially of components like device drivers that lie on the data path.
2. Discarding all the state associated with a machine results in the loss of useful side-effects occurring in that execution cycle.

High performance microreboots are achieved by taking a snapshot of the component after it has booted and initialized. The shard is modified to snapshot itself, at a time when it is ready to service requests, before these services are offered over any external interfaces. Figure 3.2 depicts the typical snapshot and rollback cycle. Lightweight snapshots rely on a hypervisor based copy-on-write mechanism to track pages that are modified.

Maintaining state across VM restarts allow services to track open connections and in-flight responses, and for storing system wide configuration data. We allow shards to persist state by specifying a region of memory as a "recovery box" [4], i.e., a block of memory that persists across rollbacks. Shards are modified to store long lived state in these areas of memory and check and restore that state immediately following a rollback.

Further implementation details about snapshots, methods to protect the stored state against attacks and the restart policies of components can be found in [15].

3.4 Deployment Scenarios

3.4.1 Public Clouds

Public Clouds, like Amazon Web Services (AWS), tightly pack a large number of virtual machines controlled by a single administrative toolstack on a single physical machine. Partitioning the platform into shards limits the risks caused by sharing resources with several Internet-exposed, non-professionally-managed, potentially-vulnerable VMs. Furthermore, judiciously microrebooting shared shards can reduce the risk with an acceptable loss of performance. Shards hosting device drivers can be independently restarted, allowing for in-place driver updates reducing the window of exposure in the face of newly discovered vulnerabilities.

3.4.2 Resource Partitioning in Private Clouds

Private clouds are often administered by more sophisticated users wanting to manage several machines as well as their interactions and the I/O path themselves. Our model supports coarse granularity resource partitioning, where each user is assigned dedicated hardware resources, either by hardware virtualization techniques like SR-IOV, or through direct device assignment, and a personal collection of shards to manage it. A private toolstack with the shards' privileges delegated to it ensures that these hardware resources are only shared amongst VMs belonging to the user. Users are allowed to virtualize their slice of hardware however they wish, dynamically creating and destroying VMs, with resource usage quotas enforced by the virtualization platform.

Chapter 4

Xen Architecture

The virtualization platform must isolate and schedule VMs, co-ordinate and regulate resource sharing, and provide a common hardware abstraction to all the VMs. It is also responsible for managing and sharing actual hardware devices.

Xen is a microkernel based hypervisor that takes a minimalistic approach to providing all this functionality. The hypervisor creates a usually Linux based, special control VM, Dom0, to manage the hardware and provide a virtualized I/O path. While Xen retains control of the serial controller and the interrupt controllers, Dom0 takes control the PCI bus, along with attached peripherals like the network and disk controllers, and is responsible for the policy determining interrupt routing to VMs. Other hypervisors are laid out differently, with KVM turning Linux into a monolithic hypervisor while VMware ESX includes device drivers in the hypervisor itself.

This division of responsibility prevents duplication of functionality and encourages code re-use leading to a significantly smaller codebase. Managing devices outside the hypervisor also reduces the privileges available to the notoriously error-prone device driver code [47], reducing the attack surface of the system.

In addition to managing the hardware, Dom0 also hosts the management toolstack responsible for creating and destroying new virtual machines. Xen relaxes normal VM isolation boundaries allowing Dom0 arbitrary access to guest VM memory, required for initialisation of guest page tables and boot time parameters. As a result, a compromise of Dom0 compromises the security of all the hosted machines and it has to be considered as part of the TCB of the entire system.

4.1 Communication Interfaces

Xen provides several communication mechanisms to VMs, the most basic being the hypercall. A hypercall is simply a software trap from a VM to the hypervisor, analogous to control flow transfer using syscalls in conventional OSes. Higher level abstractions for sharing memory and signalling between VMs are built using these hypercalls.

The existence of only around 40 hypercalls narrows the interface between VMs and the hypervisor, especially in contrast to the hundreds of system calls that allow entry to the kernel. These narrow interfaces make interactions with the hypervisor easier to analyse and audit, and also reduce the attack surface significantly [18].

Unfortunately, as more and more features have been added to Xen, the width of these interfaces has broadened with a single hypercall sometimes responsible for handling dozens of sub-operations. Operations like page faults and debug interrupts also trap to the hypervisor, further increasing the number of interfaces between guest VMs and the hypervisor.

Hypervisor exploits are not the only way to compromise the TCB of the system; Dom0 usually has a much broader interface with other guest VMs consisting of shared memory pages for virtual consoles and XenStore, event channels, and abstract devices exposed via split drivers.

4.2 Event Channels

Event Channels are a type of signalling mechanism, devoid of any associated data, primarily used for inter-VM communication, or receiving upcalls from the hypervisor. Uni-directional event channels, or VIRQs, are primarily used as a virtualized interrupt delivery mechanism by the hypervisor, which intercepts and forwards real hardware interrupts to the appropriate guest VM. Bi-directional event channels are used for notifications between VMs, like between the two halves of split drivers.

4.3 Shared Memory and I/O Rings

While VM isolation is one of the major responsibilities of a hypervisor, performance and efficiency reasons dictate that a VM may sometimes wish to share its memory with another VM.

A privileged VM may directly map the memory of another VM using a hypercall. Dom0 tools such as the VM builder, XenStore, the virtual console daemon and device emulation via QEMU all directly map the target VM's memory during VM creation.

Grant tables [17] are a more elegant, page-level granularity method of sharing data between specific non-privileged VMs. A VM can export its own pages via an access control list called the grant table, maintained by the hypervisor. Grant references act as capabilities and are passed to other VMs, whose use of them is audited against the grant table by the hypervisor.

I/O rings are bi-directional channels for asynchronous inter-VM data transfer. They rely on shared memory for data transfer and event channels for notifications. The VMs act like producer-consumer pairs, placing both requests and responses on the same ring, and independently notifying each other with event channel notifications. In an effort to be as generalized and efficient as possible, all policy is left to the users of the I/O rings, leaving the potential for malicious or malformed data to be injected via this vector.

4.4 XenStore

XenStore is a hierarchical key-value store that acts as a system-wide registry and naming service for VMs. Since both private and shared state is stored, access control policies restrict the access of a non-privileged VM to data that it owns or is explicitly authorized to access. XenStore provides a "watch" mechanism, which notifies registered listeners of any modification to particular keys in the store. As a result, XenStore is used extensively by both the toolstack and the device drivers for inter-VM synchronization or to pass small amounts of data between different VMs.

XenStore is started before any guests are created, since it contains data needed for them to boot. All VMs, including Dom0, set up an I/O ring during bootup for

XenStore communication. Since XenStore is required in the creation and bootup process, it does not use grant tables for memory sharing, but instead relies on Dom0 privileges to directly map the I/O ring for all the VMs.

Being the primary IPC mechanism between VMs, XenStore conflates several different responsibilities. VMs may need to watch or modify keys of other VMs to trigger actions such as device setup. The simplicity of the interfaces with VMs notwithstanding, this shared internal state makes it complex and vulnerable to DoS attacks if a single VM monopolizes these resources [14].

4.5 Device Drivers

Xen delegates the control of PCI peripherals, such as network and disk controllers, to Dom0, which is responsible for exposing a set of abstract devices to guest VMs. These devices may either be virtualized, passed through, or emulated.

4.5.1 Virtualized Devices

Xen uses a split driver model for high-performance, paravirtualized network and block drivers [17]. The backends, normally resident in Dom0 and interacting with the underlying hardware via system calls to the actual device drivers, expose a hardware independent, abstract device which the frontends, that act as virtual devices in guest VMs, can connect to.

Frontends and backends communicate using I/O rings. The initial negotiation is done via XenStore: a frontend driver allocates a shared page of memory and passes a grant reference and an event channel to the backend driver. The backend driver watches for this entry and establishes communication with the frontend when it appears. Data is transferred directly between the frontend and backend; Xen's only involvement is enforcing access control, using grant tables for shared memory and passing signals for synchronization.

4.5.2 Emulated Devices

Unmodified commodity OSes, on the other hand, expect to run on a standard platform. This is provided by a device emulation layer, which, in Xen, is a per-guest QEMU [6] process. QEMU emulates the BIOS, serial ports and block and network controllers for guest VMs to use. It runs in Dom0, servicing requests from the hypervisor, and has privilege to map any page of the guest VM's memory in order to emulate DMA operations. Each QEMU process can instead be hosted in its own VM by using stub domains [49].

4.5.3 Device Passthrough

Xen also allows VMs other than Dom0 to directly interface with the hardware via direct device assignment. Dom0 virtualizes the PCI bus, using a split-driver, to proxy PCI configuration space I/O and interrupt assignment requests from the guest VM to the PCI bus controller. Device specific operations are handled directly by the guest, without any involvement from Dom0 or the hypervisor.

4.6 The Toolstack

The toolstack provides administrative functions for the management of VMs. It is responsible for creating and destroying VMs, as well as managing the resources of the system. Resource allocation policies enforced by the hypervisor are applied by administrative users using the toolstack. It requires Dom0 privileges to map guest memory during VM creation to set up the guest page tables and the "start info" page, which contains parameters that guests require to interact with XenStore and the virtualized console. It registers newly created guests with XenStore, and allows access to the guest's virtualized console.

Chapter 5

Implementation

Xoar partitions the control VM of the Xen platform in line with the design described in Chapter 3. Retrofitting such a partitioning scheme onto an existing system entails considerations that would not be factors if designing a new system from the ground up. Maintaining compatibility with existing tools and virtual machines requires that the external interfaces and protocols of the system remain unchanged, often requiring more complex workarounds than would otherwise be necessary.

Xoar attempts to remain faithful to the original design, while simultaneously minimising the engineering effort involved in its choice of fault lines for creating shards, as described in Section 5.1. Figure 5.1 shows the overall architecture of Xoar, while Table 5.1 describes the set of shards in our decomposition of Dom0. While it is not the only possible decomposition, it satisfies our design goals without requiring an extensive re-engineering of Xen.

The remainder of the chapter describes the interaction between these components, and some of the challenges faced during the implementation.

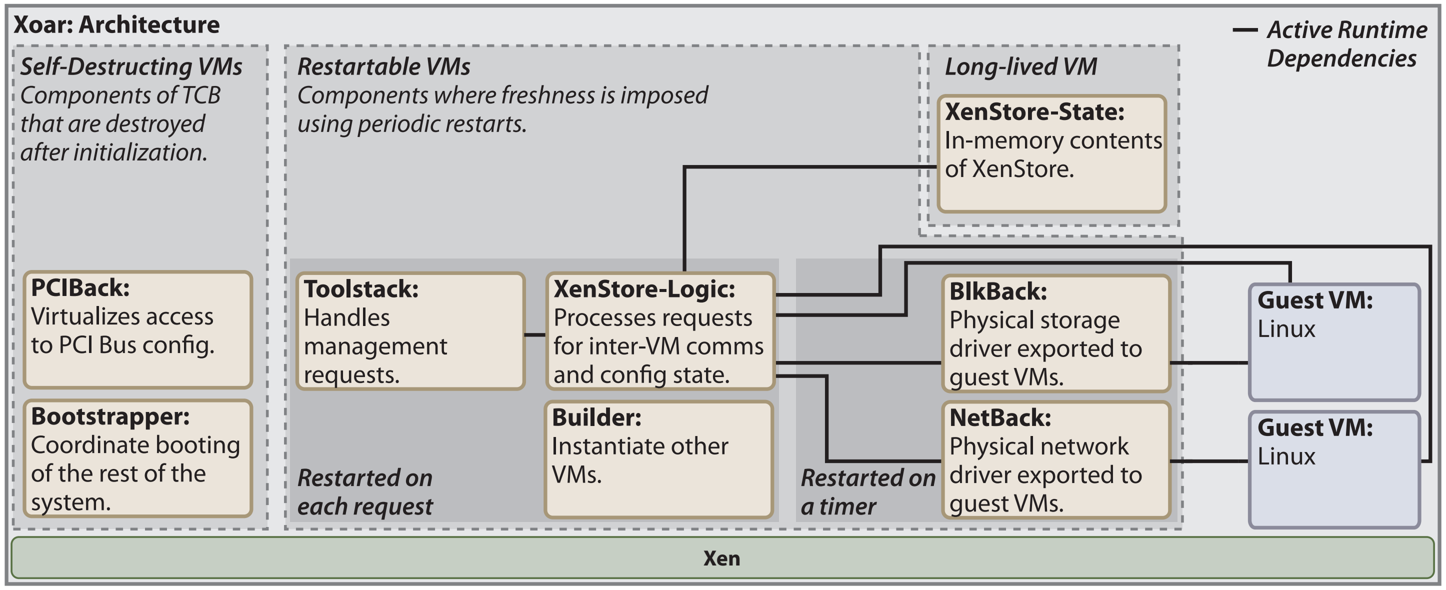
Xoar: Architecture
Active Runtime Dependencies
Self-Destructing VMs
Components of TCB that are destroyed after initialization.
PCIBack: Virtualizes access to PCI Bus config.
Bootstrapper: Coordinate booting of the rest of the system.
Restartable VMs
Components where freshness is imposed using periodic restarts.
Toolstack: Handles management requests.
XenStore-Logic: Processes requests for inter-VM comms and config state.
Builder: Instantiate other VMs.
Restarted on each request
Long-lived VM
XenStore-State: In-memory contents of XenStore.
BlkBack: Physical storage driver exported to guest VMs.
NetBack: Physical network driver exported to guest VMs.
Restarted on a timer
Guest VM: Linux
Guest VM: Linux
Xen

Figure 5.1: Architecture of Xoar

5.1 Xoar Components

The division of shards in Xoar conforms to the design goals of Chapter 3; we reduce components into minimal, loosely coupled units of functionality, while obeying the principle of least privilege. As self-contained units, they have a low degree of sharing and inter-VM communication (IVC), and can be restarted independently and without affecting other shards. Existing software and interfaces not conflicting with our design are reused to aid development and ease future maintenance.

Virtualized devices mimic physical resources closely in an attempt to offer a familiar abstraction to guest VMs, making them ideal shards. Despite the lack of toolstack support, architectural support for hosting device backends in dedicated driver domains rather than Dom0 exists, reducing the development effort significantly.

PCIBack, NetBack and BlkBack represent virtualized abstractions of the PCI bus, and network and block controllers. PCIBack virtualizes the physical PCI bus, while NetBack and BlkBack are driver domains exposing the required device backends for guest VMs. Further divison, like separating device setup from the data path yields no isolation benefits, since both components need to be shared simultaneously. Further, it would add a significant amount of IVC, thereby conflicting with the design goals, and would require extensive modifications. Similarly, the serial controller is represented by a shard that virtualizes the console for other VMs. Further details about virtualizing these hardware devices are discussed in Section 5.3, Section 5.4 and Section 5.5.

Since different aspects of the guest VM creation process require differing sets of privileges, placing them in the same shard violates our goal of reducing privilege. These operations can largely be divided into two groups – those that need access to the guest's memory to set up page tables, the start-info and the shared-info page, *etc.*, and those that require access to XenStore to write entries necessary for the guest VM. We break functionality apart along the lines of least privilege, which yields the Builder, a specialized and privileged shard responsible for the hypervisor and guest memory related operations necessary when creating a VM, and the Toolstack, a shard containing a management toolstack. While the Builder could further be divided into components for sub-operations like loading the kernel image, setting up the page tables, *etc.*, these would all need to run at the same privilege level and would incur high synchronization costs, violating our goal of reduced sharing. The Builder responds to build requests issued by a Toolstack, which, in turn, communicates with the XenStore shard to perform the rest of the configuration and setup process.

While a Toolstack exposes a lot of VM management functionality in a single shard, it is only explicitly privileged for either VMs delegated to it, or those created by it. As a result, even though further disaggregation of this component is supported by our model, it would likely unnecessarily reduce performance and increase overheads.

XenStore is both highly shared *and* privileged, a good fit for a shard. In its role as a system wide registry, it maintains state for other components in the system, and cannot be restarted trivially. We divide XenStore into a pair of components: the XenStore-State shard which is long-lived and contains all the XenStore data, and the XenStore-Logic shard which is stateless and restartable. XenStore-Logic restores this data from the XenStore-State after every restart, communicating over a single, narrow, key-value based communication protocol. Techniques to further reduce the privilege of XenStore are discussed in Section 5.6.

Component	**Privileged**	**Lifetime**	**OS**	**Parent**	**Depends on**	**Functionality**
Bootstrapper	Y	Boot Up	nanOS	Xen	-	Instantiate boot shards
XenStore	N	Forever (R)	miniOS	Bootstrapper	-	System configuration registry
Console Manager	N	Forever	Linux	Bootstrapper	XenStore	Expose physical console as virtual consoles to VMs
Builder	Y	Forever (R)	nanOS	Bootstrapper	XenStore	Instantiate non-boot VMs
PCIBack	Y	Boot Up	Linux	Bootstrapper	XenStore Builder Console Manager	Initialize hardware and PCI bus, pass through PCI devices, and expose virtual PCI config space
NetBack	N	Forever (R)	Linux	PCIBack	XenStore Console Manager	Expose physical network device as virtual devices to VMs
BlkBack	N	Forever (R)	Linux	PCIBack	XenStore Console Manager	Expose physical block device as virtual devices to VMs
Toolstack	N	Forever (R)	Linux	Bootstrapper	XenStore Builder Console Manager	Admin toolstack to manage VMs
QemuVM	N	Guest VM	miniOS	Toolstack	XenStore NetBack BlkBack	Device emulation for a single guest VM

Table 5.1: Components of Xoar (Restartable components are denoted by an “(R)” in the lifetime column)

5.2 System Boot

In a traditional Xen system, the boot process is straightforward: the hypervisor creates Dom0 during boot-up, which proceeds to initialise hardware and during PCI bus enumeration, brings up devices and their associated backend drivers. XenStore and the console daemon are started, often with boot scripts, before any guest machines are created.

On the other hand, a disaggregated system needs to consider the dependencies between components when structuring the boot order. All shards require XenStore to be running; the Console Manager needs to precede any VM requiring the virtual console; the device backends require the hardware state to already be initialized.

In Xoar, Xen first creates the Bootstrapper, which orchestrates the booting of the other core shards. XenStore is started first, as it is required by all other components, followed by the Console Manager to provide console services for the rest of the system. The Builder, which is responsible for creating VMs, must be started before PCIBack. PCIBack is the closest analogy that Xoar has to Xen's Dom0 and is responsible for hardware initialization and PCI bus enumeration. PCIBack uses `udev` [31] rules to automatically request the Builder to create instances of NetBack and BlkBack for each network and disk controller on the system. The last step is the creation of a configurable number of toolstacks.

Once booted, the toolstacks can request the Builder to create guest VMs. To avoid having the privileged Builder parse user-provided data, like kernels and file systems, it only builds VMs from a library of known good images. If a guest needs to run its own kernel, the Builder instantiates a VM with a special bootloader, which loads the user's kernel from within the guest VM.

5.3 PCI: A Shared Bus

PCIBack controls the PCI bus and manages interrupt routing for peripheral devices. Although driver VMs have direct access to the peripherals themselves, the shared nature of the PCI configuration space mandates that a single component multiplex all accesses to it.

The configuration registers are used during device initialisation, but once the device is running, there is no further communication between the PCI split driver frontends and backends under normal operating conditions. As a result, once steady state is achieved, we can remove PCIBack from the TCB entirely, either by de-privileging or destroying it, and reduce the privileged components in the system.

In our system, we are able to remove PCIBack because there is no further interaction with shared PCI state. Hardware virtualization techniques like SR-IOV [32] allow the creation of virtualized devices, where the multiplexing is performed in hardware, thereby obviating the need for driver domains. However, provisioning new virtual devices on the fly, requires a *persistent* shard to assign interrupts and multiplex accesses to the PCI configuration space. Ironically, although appearing to reduce the amount of sharing in the system, such techniques may increase the number of shared, trusted components.

5.4 Driver Domains: NetBack and BlkBack

Driver domains, such as NetBack and BlkBack, directly access PCI peripherals like the NIC and disk controller using PCI device passthrough, and rely on existing driver support in Linux to interface with the hardware. Each NetBack or BlkBack virtualizes exactly one network or block controller, hosting the relevant device driver and the virtualized backend required by guests. During VM creation, the Toolstack links a guest VM to the selected driver domain by writing the appropriate frontend and backend XenStore entries.

Separating BlkBack from the Toolstack causes some problems, because the existing Toolstack mounts disk based images of guest VMs as loopback devices using `blktap` for use by the backend drivers. After splitting BlkBack and the Toolstack, the disk images need to be mounted in BlkBack. Similarly, administrators create new files or partitions from the Toolstack to back new guest VMs. In Xoar, BlkBack runs a lightweight daemon that acts as a proxy for requests of the Toolstacks.

5.5 Sharing the Console: Console Manager

Xen controls the serial port itself, sharing the console with Dom0 using shared memory and a dedicated VIRQ. Dom0 virtualizes this console for other guest domains using I/O rings and a user space console daemon (`xenconsoled`). The Console Manager is started before any other Linux guest VMs, and hosts this user space daemon to expose a virtualized serial console to all other guest VMs. Unfortunately, Linux enumerates the PCI bus taking control of all unassigned devices and PCI interrupt routing during system boot, leaving no hardware devices for PCIBack to virtualize. The Console Manager modifies the boot process to skip device enumeration and jump to I/O-Port initialisation to avoid this problem. Like XenStore, the Console Manager is deprivileged using the techniques in Section 5.6.

5.6 Deprivileging Tools and Multiple Toolstacks

XenStore and the Console Manager require Dom0-like privileges to forcibly map shared memory, since they are required before the guest VM can set up its grant table mappings. The Builder adds a step to the regular VM creation code – to automatically create grant table entries for this shared memory, allowing these tools to use grant tables and function without any special privileges.

Each Toolstack hosts a management toolstack, which is built on top of the xenlight library (`libxl`). A Toolstack creates guest virtual machines by passing parameters to the Builder. Each Toolstack instance is assigned VM-management privileges, such as starting or stopping the guest VM, or modifying its resource allocation by the Builder, for all VMs that it requests built. A toolstack can only manage these VMs, and an attempt to manage any other guests is blocked by the hypervisor.

Qemu stub domains for HVM guests require limited privileges over the memory of the guest for DMA purposes, leading to the addition of a flag allowing a VM to be specified as privileged for another VM. In a similar vein, we add a flag marking the parent Toolstack for every guest VM, which is set during VM creation. Privileged VM management hypercalls are audited against this flag, and only those originating from the parent Toolstack are executed.

A Toolstack can only use shards that have been delegated to it as shared resource providers for VMs that it requests built. An attempt to use a VM that is not a shard, or an undelegated shard, as say a NetBack, for a new guest VM would fail.

Shared resource providers require the co-operation of the hypervisor to use I/O

Rings for sharing data and event channels for IVC with consuming guest VMs. Requests to share memory between VMs and event channel operations are blocked if at least one of VMs is not a shard, or if the guest VM is not delegated to use that particular shard.

5.7 Developing with Minimal OSes

Bootstrapper and Builder are built on top of nanOS, a small, single-threaded, and lightweight kernel, explicitly designed to have the minimum functionality needed for VM creation. The small size and complexity of these components leave them well within the realm of static analysis techniques, which could be used to verify their correctness. XenStore, on the other hand, is built on top of miniOS, a more feature complete OS distributed as a stub domain environment with Xen, because of the necessity of a multithreading environment.

Determining the correct size of OS is hard, with a fundamental tension between functionality and the ease of use. Keeping nanOS so rigidly simple introduces a set of development challenges, especially in cases involving IVC. For example, the XenStore watch mechanism, described earlier and used for inter-VM IPC, is naturally described in a multi-threaded model, but significantly harder in single-threaded environments. However, as the only privileged VM in Xoar is based on nanOS, we felt that the improvement in security due to reduced complexity was a worthwhile trade-off.

5.8 Implicit Assumptions about Dom0

Although the design of Xen does not mandate that all service components must live in Dom0, it is implicitly assumed in several instances, with various access-control checks made by comparing the caller's ID with a hard-coded 0, the ID for Dom0.

The hypervisor sets up MMIO and I/O-port privileges for access to the console and peripherals, and delivers signals for the console to Dom0. In Xoar, these need to be mapped to the correct domains, with Console Manager requiring signals and console I/O-port access, and PCIBack requiring the remaining I/O-port and MMIO privileges, along with access to the PCI bus. Also, the hypervisor assumes that a Dom0 failure is critical and reboots the entire system, which needs to be modified to allow the Bootstrapper to complete execution and quit.

Many tools assume that they are running in Dom0, co-located with the driver backends. Various paths in XenStore are hard-coded to be in Dom0's directory. The Toolstack also expects to be able to manipulate the files that contain disk images which is solved by proxying requests as discussed in Section 5.4.

5.9 Implementation Effort and Future Maintainability

The control VM provides various different services required by guest VMs including low-level hardware access and multiplexing, the result being that this division involved challenging OS-level restructuring requiring a significant amount of implementation effort.

Purely decomposing the control VM into shards only needs to lightly modify the hypervisor to account for the lack of co-location of service components. Additional security features such as the snapshot and rollback mechanism, however, require more substantial modifications to the hypervisor.

Building the appropriate shards consumed a large fraction of the overall development; for instance, both XenStore-Logic and XenStore-State, and the Builder and nanOS are built from scratch.

We believe that a significant proportion of this restructuring is a one-time change, and once integrated would reduce the effort required to disaggregate future versions of Xen substantially.

Chapter 6

Evaluation

Xoar significantly re-engineers the hypervisor, the paravirtualized Linux kernel (pvops) and the toolstack of the Xen platform to meet its goal of partitioning the monolithic control VM; however, to be practically useful it should show tangible security benefits without imposing prohibitive overheads. We justify this engineering effort by comparing the performance and security of our prototype against the existing version of Xen, and show that it is a viable alternative for commercial deployments.

6.1 Performance Evaluation

The performance of Xoar is evaluated against a stock Xen Dom0 in terms of memory overhead, I/O throughput, boot time and overall system performance. Each shard in Xoar runs with a single virtual CPU; in stock Xen Dom0 runs with 2 virtual CPUs, the configuration used in the XenServer [11] commercial Xen platform. All figures are the average of three runs, with 95% confidence intervals shown where appropriate.

Our test system was a Dell Precision T3500 server, with a quad core, 2.67GHz Intel Xeon W3520 processor, 4GB of RAM, a Tigon 3 Gigabit Ethernet card, and an Intel 82801JIR SATA controller with a Western Digital WD3200AAKS-75L9A0 320GB 7200 RPM disk. VMX, EPT and IOMMU virtualization are en-

Component	Memory	Component	Memory
XenStore-Logic	32MB	XenStore-State	32MB
Console Manager	128MB	PCIBack	256MB
NetBack	128MB	BlkBack	128MB
Builder	64MB	Toolstack	128MB

Table 6.1: Memory Consumption of Individual Shards

abled. We use Xen 4.1.0 and Linux 2.6.31[1] pvops kernels for the tests. Identical guests running an Ubuntu 10.04 system, configured with 2 VCPUs, 1 GB of RAM and a 15 GB virtual disk are used on both systems. For network tests, the system is connected directly to another system with an Intel 82567LF-2 Gigabit network controller.

6.1.1 Memory Overhead

Table 6.1 shows the memory requirements of each of the components in Xoar. Systems with multiple network or disk controllers can have several instances of the NetBack and BlkBack shards. Also, since users can select the shards to run, there is no single figure for total memory consumption. In commercial hosting solutions, console access is largely absent rendering the Console Manager redundant. Similarly, PCIBack can be destroyed if hardware virtualization is not used. As a result, the memory requirements range from 512 MB to 896 MB, assuming a single network and block controller, representing a saving of 30% to an overhead of 20% on the default 750MB Dom0 configuration used by XenServer. All performance tests compare a complete configuration of Xoar with a standard Dom0 Xen configuration.

6.1.2 I/O Performance

Disk performance is tested using Postmark, with VMs' virtual disks backed by files on local disk. Figure 6.1 shows the results of running a typical configuration with default parameters.

Network performance is tested by fetching a 512 MB and a 2 GB file across a Gigabit LAN using `wget`, and writing it either to disk, or to `/dev/null` to

[1]Hardware issues forced us to use a 2.6.32 kernel for some of the components.

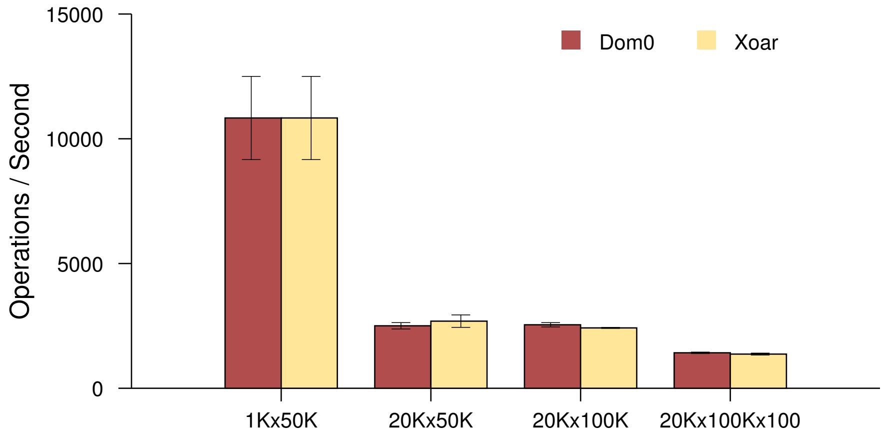

Figure 6.1: Disk performance using Postmark

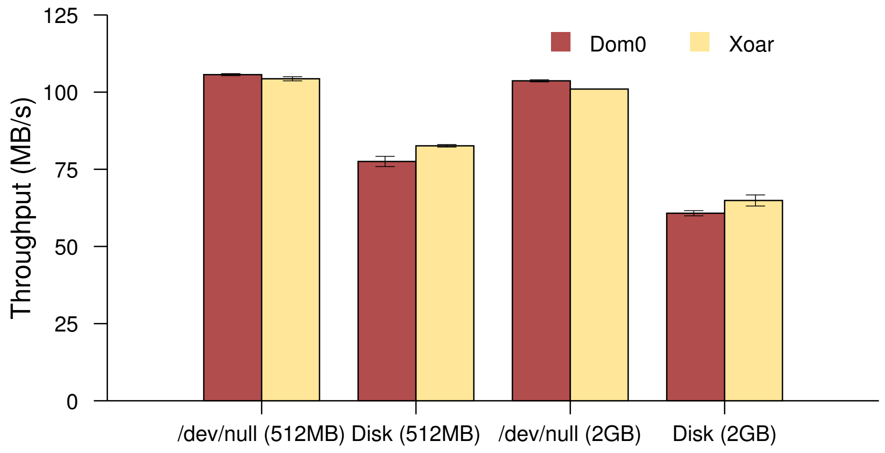

Figure 6.2: Network performance with `wget`

eliminate performance artifacts due to disk performance. Figure 6.2 shows these results.

Overall, disk throughput is more or less unchanged, and network throughput is down by 1-2.5%. The combined throughput of data coming from the network onto the disk is *up* by 6.5%; we believe this is caused by the performance isolation of running the disk and network drivers in separate VMs.

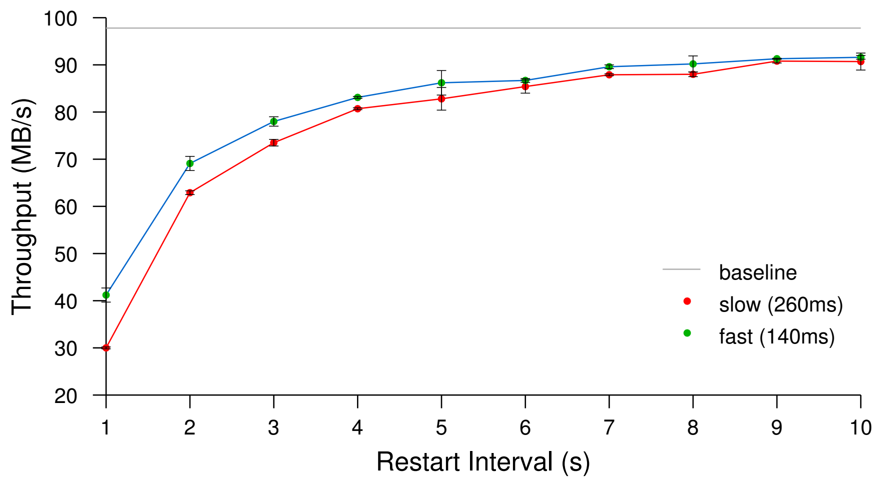

Figure 6.3: Throughput with a Restarting NetBack

To measure the effect of microrebooting driver VMs, we ran the 2GB `wget` to `/dev/null` while restarting NetBack at intervals between 1s and 10s. Two different optimizations for fast microreboots are shown.

In the first (marked as "slow" in Figure 6.3), the device hardware state is left untouched during reboots; in the second ("fast"), some configuration data that would normally be renegotiated via XenStore is persisted. In "slow" restarts the device downtime is around 260ms, as measured from between the time the device is suspended to the time it responds to network traffic again. The optimisations used in the "fast" restart reduce this downtime to around 140ms.

Resetting every 10 seconds causes an 8% drop in throughput, as `wget`'s TCP connections respond to the breaks in connectivity. Increasing to every second gives a 58% drop. Increasing the interval beyond 10s makes very little difference to throughput. The faster recovery gives a noticeable benefit for very frequent reboots but is worth less than 1% for 10-second reboots.

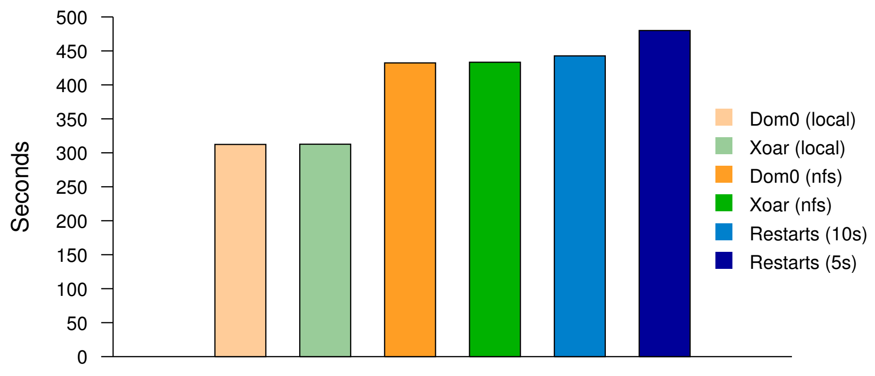

Figure 6.4: Kernel Build: Local and Remote NFS

6.1.3 System Boot

Boot time is usually measured from the time a system is powered on till the time it accepts user input. Unlike Dom0, the Console Manager in Xoar does not have to negotiate with other hardware before displaying a login prompt and boots 1.5 times faster.

	Dom0	**Xoar**	**Speedup**
Console	38.9s	25.9s	1.5
`ping`	42.2s	36.6s	1.15

Table 6.2: Comparison of Boot Times

In the interest of fairness, we also tested waiting for hardware negotiation to finish and the system to start responding to external `ping` requests, and still found Xoar to be 1.15 times faster (see Table 6.2). We believe that the improved boot time is a result of parallel booting that can occur due to the compartmentalisation of components.

6.1.4 Real-world Benchmarks

Figure 6.4 compares the time taken to build a Linux kernel, both in stock Xen and Xoar, off a local ext3 volume as well as an NFS mount. The overhead added by Xoar is much less than 1%.

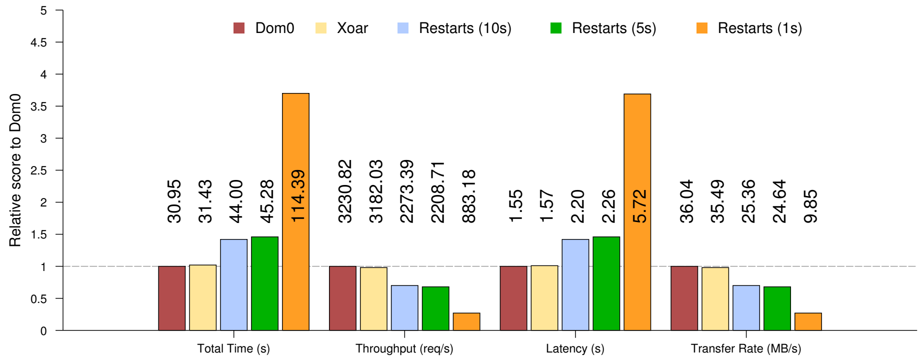

Figure 6.5: Apache Benchmark:Regular and with NetBack Restarts

The *Apache Benchmark* is used to gauge the performance of an Apache web server serving a static web page to many simultaneous clients. Figure 6.5 shows the results of this test against Dom0, Xoar, and Xoar with network driver restarts at 10, 5, and 1 second intervals. Performance decreases non-uniformly with the frequency of the restarts: an increase in restart interval from 5 to 10 seconds yields barely any performance improvements, while changing the interval from 5 seconds to 1 second introduces a significant performance loss.

Dropped packets and network timeouts cause a small number of requests to experience very long completion times – for example, for Dom0 and Xoar, the longest packet took only 8-9ms, but with restarts, the values range from 3000ms (at 5 and 10 seconds) to 7000ms (at 1 second). As a result, the longest request interval is not shown in the figure.

Overall the performance impact of disaggregation is quite low. This is largely because driver VMs do not lengthen the data path between guests and the hardware. Rather than communicating with driver backends in Dom0, the guest VM communicates directly with the backends located in NetBack or BlkBack, which drive the hardware. The overhead of driver restarts is noticeable, but can be tuned by the administrator to best match the desired combination of security and performance.

6.2 Security Evaluation

Xoar restructures the Xen platform so that rather than a Linux shard, only a single, small nanOS shard has the privileges required to arbitrarily access a guest's memory; as a result, Xen's TCB is reduced from Linux's 7.6 million (400,000 compiled) lines of code to 13,000 (8,000 compiled) lines of code, both on top of the Xen hypervisor's 280,000 (70,000 compiled) lines of code. While still significantly larger than microkernels [35], the Xen codebase is inflated by the need to support multiple architectures and guest types as demanded by enterprise.

In addition to echoing the anthemic "small is secure" argument, Xoar offers several other benefits over a stock Xen platform. Shards lessen the consequences of an attack by limiting the attacker to the privilege level of the compromised component. An attacker exploiting a vulnerability in say, the emulated device model, will now have the full privileges of the QemuVM, rather than Dom0 privileges and has no rights over any other VM.

An attacker compromising a driver domain shard gains access only to the degree that other VMs trust that shard – compromising NetBack would allow intercepting the network traffic of another VM relying on the same NetBack, but not reading or writing its memory.

Buggy, outdated and vulnerable device drivers often continue to be used, despite the inherent security risks, because of the downtime and costs associated with upgrading a single driver in a monolithic control VM. Shards that can independently be restarted without disturbing the entire system allow in-place driver upgrades with minimal downtime, easing the process of patching security vulnerabilities.

Xoar allows the user much finer control on the exposure of each VM. Configurations where no VMs share any shards, or sharing is restricted to mutually trusting VMs are possible, especially on hosts possessing enough surplus hardware resources. In such configurations, the only shared component between two VMs is the hypervisor itself.

Oftentimes, however, either such excess hardware resources are not available or undermine the low-cost, densely packed VM philosophy that forms the underpinnings of enterprise cloud deployments. Frequent microreboots, to reduce the temporal attack surface, and better auditing tools to *ex post facto* determine the level of exposure and the degree of sharing shards with compromised VMs help secure users in scenarios where some sharing is inevitable.

Xoar does not favour a particular configuration allowing users to select the one best suited to their needs – from tightly packed deployments with all resources shared, to completely independent deployments that only share the hypervisor, while increasing the security or transparency in all the cases.

6.2.1 Known Attacks

Of the attacks discussed in Chapter 2, Xoar entirely contains the 7 device emulation attacks, as the device emulation shard has no rights over any VM except the one the attack came from. The 6 attacks on the virtualized device layer and the 1 attack on the toolstack would yield control only over those VMs that shared the same BlkBack, NetBack and Toolstack components.

There were 2 exploits on debug registers which can be mitigated by deprivileging guests. This could be done on either Xen or Xoar. There were 2 exploits on XenStore write access, which was caused by a bug in XenStore itself. The version of Xen we used already had these bugs fixed. We would currently not be able to protect against the hypervisor exploit.

Chapter 7

Discussion and Future Work

Software partitioning has been explored in a variety of contexts before, with microkernels long advocating message passing based systems with small TCBs [34, 35]. However, they remain largely restricted to the domain of embedded devices with relatively small and focused development teams [29]. Attempts at application-level partitioning have demonstrated real benefits in terms of securing sensitive data, while simultaneously demonstrating challenges in implementation and concerns about maintenance [7, 9, 26, 40], primarily due to the mutability of application interfaces.

Isolating the largely independent, shared components running in the control VM above the hypervisor did not exhibit these issues to nearly the same degree as these components typically are either drivers or application code exposing their interfaces to either the underlying hardware or to dependent guests. Representing such services as independent shards was a surprisingly natural fit.

While tempting to attribute this as a general property of virtualization platforms, we also think that it was particularly applicable to the architecture of Xen. Although implemented as a monolithic TCB, several components were designed to support further compartmentalisation; self-contained units with little to no shared state and clear, narrow communication interfaces.

We believe the same is applicable to Hyper-V, which is architecturally similar to Xen. In contrast, in KVM [28] the Linux kernel is converted to a hypervisor, with the entire management toolstack and device emulation framework hosted in a single QEMU process. Linux host based device drivers interact with the underlying hardware. We believe that the high degree of coupling and significant shared state between the components would make such as aggressive disaggregation of KVM extremely hard, more akin to converting Linux into a microkernel.

7.1 Future Work

Xoar disaggregates Dom0 into a set of isolated shards, reducing the TCB of Xen to just the hypervisor and the Builder. However, the hypervisor itself remains unpartitioned, with all the code running with heightened privileges. While operations like guest page table updates, I/O-port management, trap and emulate handlers, *etc.*, require these capabilities, operations like domain management, profiling and tracing and so on function correctly even when run in a lower privileged hardware protection domain.

We hope to further reduce the TCB of virtualization platforms by splitting the hypervisor into a privileged and non-privileged component, which run in different hardware protection rings. The privileged component would receive non-privileged hypercalls and transfer them to the non-privileged component, similar to IPC mechanisms for handling syscalls in microkernels [2, 34].

The mechanism of rebooting components that automatically renegotiate existing connections allow many parts of the virtualization platform to be upgraded in place. An old component can be shut down gracefully, and a new, upgraded one brought up in its place with minor modifications to certain XenStore keys. Unfortunately, these same techniques are not applicable to long-lived components with persistent state like XenStore, or the hypervisor itself.

XenStore could potentially be restarted by persisting its state to disk, and checking and recovering that state on restart. Restarting Xen under executing VMs, however, is more challenging. We would like to explore techniques like those in ReHype [33], but using *controlled* reboots to safely replace Xen, allowing the complete virtualization platform to be upgraded and restarted without disturbing the hosted VMs.

Chapter 8

Conclusion

Xoar is an architectural change to the virtualization platform that retrofits microkernel like isolation properties to the Xen hypervisor without sacrificing any existing functionality, while simultaneously remaining fully compatible with existing guest virtual machines.

Enterprise demand has spurred the growth of low-cost, centralized hosting and management of systems in the cloud, largely ignoring the security risks posed by sharing resources amongst such densely co-located, mutually untrusting systems. The virtualization layer, while designed to be small and secure, has grown out of a need to support features desired by such enterprises.

Xoar partitions the monolithic control VM of the virtualization layer into a set of isolated, least-privilege shards. It effectively restricts the scope of an attacker to the privileges associated with the compromised shard and significantly reduces the size of the TCB of the entire system. Sharing shards between guests needs to be explicitly specified, allowing users to limit their exposure and audit accesses to shared resources. By eliminating shared state between such decoupled components, Xoar is able to independently and frequently microreboot shared components to freshen their state and reduce the temporal attack surface of the system.

Providing enhanced security while maintaining the feature and performance parity needed for large-scale enterprise adoption required a considerable amount of engineering. Xoar is designed to be a drop-in replacement for the Xen platform, allowing administrators an easy migration path without any modifications to existing infrastructure. It enables unconventional deployment scenarios in private clouds, allowing users to purchase resources and decide the optimal partitioning and sharing strategies between their own guest virtual machines.

Though not insignificant, the implementation largely entailed building communication interfaces between service components and modifying the hypervisor's assumptions about their co-location. These one-time changes, once integrated, would reduce the effort of disaggregating future releases significantly. We believe that this architecture shows considerable promise, with very few downsides compared to existing virtualization platforms.

Bibliography

[1] *Department of Defense Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD. U.S. Department of Defense, December 1985. → pages 4

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986. → pages 10, 41

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM SOSP*, pages 1–14, October 2009. → pages 2

[4] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. USENIX Summer Conference*, pages 31–43, June 1992. → pages 15

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. 19th ACM SOSP*, pages 164–177, October 2003. → pages 1

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC*, pages 41–46, April 2005. → pages 20

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proc. 5th USENIX NSDI*, pages 309–322, April 2008. → pages 40

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proc. 15th SOSP*, pages 1–11, December 1995. → pages 3

[9] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *Proc. 13th USENIX Security Symposium*, pages 57–72, August 2004. → pages 40

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proc. 6th USENIX OSDI*, pages 31–44, December 2004. → pages 3, 11

[11] Citrix Systems, Inc. *Citrix XenServer 5.6 Admininistrator's Guide*. June 2010. → pages 32

[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proc. 2nd USENIX NSDI*, pages 273–286, May 2005. → pages 3

[13] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *Proc. USENIX ATC*, pages 101–114, June 2009. → pages 3

[14] Patrick Colp. [xen-devel] [announce] xen ocaml tools. http://lists.xensource.com/archives/html/xen-devel/2009-02/msg00229.html. → pages 20

[15] Patrick Colp. Eliminating the Long-Running Process: Separating Code and State. M.Sc. Thesis, 2010. → pages 16

[16] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. 5th USENIX NSDI*, pages 161–174, April 2008. → pages 3

[17] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st OASIS*, October 2004. → pages 9, 19, 20

[18] Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold Co., New York, NY, USA, 1988. → pages 18

[19] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proc. 20th ACM SOSP*, pages 163–176, October 2005. → pages 14

[20] Jakob Gorm Hansen and Eric Jul. Lithium: Virtual machine storage for the cloud. In *Proc. 1st ACM SOCC*, pages 15–26, June 2010. → pages 3

[21] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat.

Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, October 2010. → pages 3

[22] Jim Held, Jerry Batista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel Corporation, September 2006. → pages 3

[23] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proc. 11th ACM SIGOPS EW*, September 2004. → pages 5

[24] Kason Kappel, Anthony Velte, and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, 1st edition, 2010. → pages 1

[25] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM. → pages 2, 8

[26] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proc. USENIX ATC*, pages 273–284, June 2003. → pages 40

[27] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM. → pages 14

[28] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. `kvm`: the Linux virtual machine monitor. In *Proc. Linux Symposium*, pages 225–230, July 2007. → pages 40

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proc. 22nd ACM SOSP*, pages 207–220, October 2009. → pages 40

[30] Kostya Kortchinsky. Cloudburst - hacking 3D and breaking out of VMware. In *Black Hat USA*, July 2009. → pages 6

[31] Greg Kroah-Hartman. `udev`: A userspace implementation of devfs. In *Proc. Linux Symposium*, pages 263–271, July 2003. → pages 27

[32] Patrick Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. Application note 321211-002, Intel Corporation, January 2011. → pages 28

[33] Michael Le and Yuval Tamir. ReHype: Enabling VM survival across hypervisor failures. In *Proc. 7th ACM VEE*, March 2011. → pages 41

[34] Jochen Liedtke. Improving ipc by kernel design. In *In 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, 1993. → pages 40, 41

[35] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM. → pages 38, 40

[36] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. USENIX ATC*, pages 29–42, June 2001. → pages 8

[37] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual machines. In *Proc. 3rd EuroSys*, pages 41–54, March 2008. → pages 3

[38] Grzegorz Milos, Derek Gordon Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proc. USENIX ATC*, pages 1–14, June 2009. → pages 3

[39] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *Proc. 4th ACM VEE*, pages 151–160, March 2008. → pages 5, 8

[40] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proc. 12th USENIX Security Symposium*, pages 231–242, August 2003. → pages 40

[41] Joanna Rutkowska and Rafal Wojtczuk. *Qubes OS Architecture*. Version 0.3. January 2010. http://qubes-os.org/. → pages 9

[42] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Proc. 21st ACSAC*, pages 276–285, December 2005. → pages 8

[43] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group: Perspectives and perceptions. In *Proc. 15th ICFP*, pages 87–92, September 2010. → pages 8

[44] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. 21st ACM SOSP*, pages 335–350, October 2007. → pages 8

[45] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *Proc. 5th ACM VEE*, pages 121–130, March 2009. → pages 8

[46] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. 5th EuroSys*, pages 209–222, April 2010. → pages 2, 8

[47] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM. → pages 17

[48] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39(5):44–51, May 2006. → pages 2

[49] Samuel Thibault and Tim Deegan. Improving performance by embedding HPC applications in lightweight Xen domains. In *Proc. 2nd HPCVIRT*, March 2008. → pages 9, 21

[50] VMware Inc. *The VMware Reference Architecture for Stateless Virtual Desktops with VMware View 4.5*. October 2010. → pages 3

[51] VMware Inc. *vSphere Resource Management Guide*. Number EN-000317-00. May 2010. → pages 3