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

Patrick Colp[†], Mihir Nanavati[†], Jun Zhu[‡], William Aiello[†],
George Coker[*], Tim Deegan[‡], Peter Loscocco[*], and Andrew Warfield[†]

[†]Department of Computer Science, University of British Columbia
[‡]Citrix Systems R&D, [*]National Security Agency

ABSTRACT

Cloud computing uses virtualization to lease small slices of large-scale datacenter facilities to individual paying customers. These *multi-tenant* environments, on which numerous large and popular web-based applications run today, are founded on the belief that the virtualization platform is sufficiently secure to prevent breaches of isolation between different users who are co-located on the same host. Hypervisors are believed to be trustworthy in this role because of their small size and narrow interfaces.

We observe that despite the modest footprint of the hypervisor itself, these platforms have a large aggregate trusted computing base (TCB) that includes a monolithic control VM with numerous interfaces exposed to VMs. We present *Xoar*, a modified version of Xen that retrofits the modularity and isolation principles used in microkernels onto a mature virtualization platform. Xoar breaks the control VM into single-purpose components called *service VMs*. We show that this componentized abstraction brings a number of benefits: sharing of service components by guests is configurable and auditable, making exposure to risk explicit, and access to the hypervisor is restricted to the least privilege required for each component. Microrebooting components at configurable frequencies reduces the temporal attack surface of individual components. Our approach incurs little performance overhead, and does not require functionality to be sacrificed or components to be rewritten from scratch.

1. INTRODUCTION

Datacenter computing has shifted the criteria for evaluating systems design from one that prioritizes peak capacity and offered load, to one that emphasizes the efficiency with which computing is delivered [2, 5, 47, 45]. This is particularly true for cloud hosting providers, who are motivated to reduce costs and therefore to multiplex and over-subscribe their resources as much as possible while still meeting customer service level objectives (SLOs).

While the efficiency of virtualization platforms remains a primary factor in their commercial success, their administrative features and benefits have been equally important. For example, hardware failures are a fact of life for large hosting environments; such environments rely on functionality such as live VM migration [13] for planned hardware replacements as well as unexpected failures [8, 15]. Hardware diversity is also inevitable in a large hosting facility; the use of hardware emulation and unified virtual devices means that a single VM image can be hosted on hardware throughout the facility without the need for device driver upgrades within customer VMs. Administrative benefits aside, the largest reason for the success of virtualization may be that it requires little or no change to existing applications. These three factors (resource utilization, administrative features, and the support of existing software) have allowed the emergence of large-scale hosting platforms, such as those offered by Amazon and Rackspace, that customers can trust to securely isolate their hosted virtual machines from those of other tenants despite physical co-location on the same physical hardware.

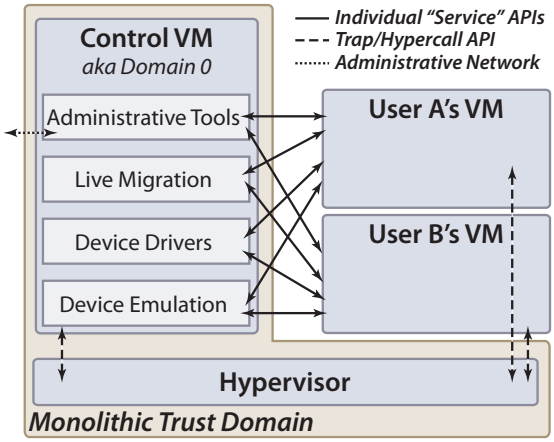

Figure 1: The control VM is often a full operating system install, has privilege similar to the hypervisor, and offers multiple services over numerous interfaces to guest VMs.

Are hypervisors worthy of this degree of trust? Proponents of virtualization claim that the small trusted computing base (TCB) and narrow interfaces afforded by a hypervisor provide strong isolation between the software components that share a host. In fact, the TCB of a mature virtualization platform is *larger* than that of a conventional server operating system. Even Type-1 hypervisors, such as Xen [4] and Hyper-V [22], rely on a privileged OS to provide additional shared services, such as drivers for physical devices, device emulation, and administrative tools. While the external interfaces to these services broaden the attack surface exposed to customer VMs, the internal interfaces *between* components within that OS are not as narrow or as carefully protected as those between components of

SOSP '11, October 23-26, 2011, Cascais, Portugal.

the hypervisor itself. This large control VM is the "elephant in the room", often ignored in discussing the security of these systems.

While TCB size may not be a direct representation of risk, the shared control VM *is* a real liability for these systems. In Xen, for instance, this control VM houses a smorgasbord of functionality: device emulation and multiplexing, system boot, administrative toolstack, etc. Each of these services is presented to multiple customer VMs over different, service-specific interfaces (see Figure 1). As these services are all part of a single monolithic TCB, a compromise of any of them places the entire platform in danger.

The history of OS development shows us how to address the problem of a large TCB: break it into smaller pieces, isolate those pieces from each other, and reduce each one to the least privilege consistent with its task [43]. However, the history of OS deployment demonstrates that "secure by design" OSes often generate larger communities of readers than developers or users. In this vein, from-scratch hypervisors [38, 40, 42] have shown that particular security properties can be achieved by rearchitecting the platform, but they do not provide the rich set of features necessary for deployment in commercial hosting environments.

The work described in this paper avoids this compromise: we address the monolithic TCB presented by the control VM *without* reducing functionality. Instead, we hold the features of a mature, deployed hypervisor as a baseline and harden the underlying TCB. Our approach is to incorporate stronger isolation for the existing components in the TCB, increasing our ability to control and reason about exposure to risk. While full functionality is necessary, it is not sufficient for commercial deployment. Our approach adds only a small amount of performance overhead compared to our starting point full-featured virtualization platform.

1.1 Contributions

The primary contribution of this paper is to perform a component-based disaggregation of a mature, broadly deployed virtualization platform in a manner that is practical to incorporate and maintain. Our work takes advantage of a number of well-established mechanisms that have been used to build secure and reliable systems: the componentization of microkernels, freshening of component state using microreboots [10], and the use of recovery boxes [3] to allow a small set of explicitly designated state to survive reboots. The insight in this work is that these techniques can be applied to an existing system along the boundaries that already exist between processes and interfaces in the control VM.

We describe the challenges of decomposing Xen's control VM into a set of nine classes of *service VMs* while maintaining functional, performance, and administrative parity. The resulting system, which we have named *Xoar*, demonstrates a number of interesting new capabilities that are not possible without disaggregation:

- **Disposable Bootstrap.** Booting the physical computer involves a great deal of complex, privileged code. Xoar isolates this functionality in special purpose service VMs and destroys these VMs before the system begins to serve users. Other Xoar components are microrebooted to known-good snapshots, allowing developers to reason about a specific software state that is ready to handle a service request.
- **Auditable Configurations.** As the dependencies between customer VMs and service VMs are explicit, Xoar is able to record a secure audit log of all configurations that the system has been placed in as configuration changes are made. We show that this log can be treated as a temporal database, enabling providers to issue forensic queries, such as asking for a list of VMs that depended on a known-vulnerable component.
- **Hardening of Critical Components.** While a core goal of our work has been to minimize the changes to source in order to make these techniques adoptable and maintainable, some critical components are worthy of additional attention. We identify *XenStore*, Xen's service for managing configuration state and inter-VM communication, as a sensitive and long-running component that is central to the security of the system. We show how isolation and microreboots allow XenStore to be rearchitected in a manner whereby an attacker must be capable of performing a stepping-stone attack across two isolated components in order to compromise the service.

We believe that Xoar represents a real improvement to the security of these important systems, in a manner that is practical to incorporate today. After briefly describing our architecture, we present a detailed design and implementation. We end by discussing the security of the system and evaluate the associated performance costs.

2. TCBS, TRUST, AND THREATS

This section describes the TCB of an enterprise virtualization platform and articulates our threat model. It concludes with a classification of relevant existing published vulnerabilities as an indication of threats that have been reported in these environments.

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]. 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" [21, 33].

Enterprise virtualization platforms, such as Xen, VMware ESX, and Hyper-V, are responsible for the isolation, scheduling, and memory management of guest VMs. Since the hypervisor runs at the highest privilege level, it forms, along with the hardware, part of the system's TCB.

Architecturally, these platforms rely on additional components. Device drivers and device emulation components manage and multiplex access to I/O hardware. Management toolstacks are required to actuate VMs running on the system. Further components provide virtual consoles, configuration state management, inter-VM communication, and so on. Commodity virtualization platforms, such as the ones mentioned above, provide all of these components in a monolithic domain of trust, either directly within the hypervisor or within a single privileged virtual machine running on it. Figure 1 illustrates an example of this organization as implemented in Xen.

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 can access its interfaces to other elements of the TCB which allows them to attempt to inject malicious requests or responses over those interfaces.

Example Attack Vectors. We analyzed the CERT vulnerability database and VMware's list of security advisories, identifying a total of 44 reported vulnerabilities in Type-1 hypervisors.[1] Of the reported Xen vulnerabilities, 23 originated from within guest VMs,

[1]There were a very large number of reports relating to Type-2 hy-

11 of which were buffer overflows allowing arbitrary code execution with elevated privileges, while the other eight were denial-of-service attacks. Classifying by attack vector showed 14 vulnerabilities in the device emulation layer, with another two in the virtualized device layer. The remainder included five in management components and only two hypervisor exploits. 21 of the 23 attacks outlined above are against service components in the control VM.

Threat Model. We assume a well-managed and professionally administered virtualization platform that restricts access to both physical resources and privileged administrative interfaces. That is, we are not concerned with the violation of guest VM security by an administrator of the virtualization service. There are business imperatives that provide incentives for good behavior on the part of hosting administrators.

There is no alignment of incentives, however, for the guests of a hosting service to trust each other, and this forms the basis of our threat model. In a multi-tenancy environment, since guests may be less than well administered and exposed to the Internet, it is prudent to assume that they may be malicious. Thus, the attacker in our model is a guest VM aiming to violate the security of another guest with whom it is sharing the underlying platform. This includes violating the data integrity or confidentiality of the target guest or exploiting the code of the guest.

While we assume that the hypervisor of the virtualization platform is trusted, we also assume that the code instantiating the functionality of the control VM *will* contain bugs that are a potential source of compromise. Note that in the case of a privileged monolithic control VM, a successful attack on any one of its many interfaces can lead to innumerable exploits against guest VMs. 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 functional components in space and time so that an exploit of one component is not sufficient to mount a successful attack against another guest or the underlying platform.

3. ARCHITECTURE OVERVIEW

Before explaining the design goals behind Xoar, it is worth providing a very high-level overview of the components, in order to help clarify the complexities of the control plane in a modern hypervisor and to establish some of the Xen-specific terminology that is used throughout the remainder of the paper. While our implementation is based on Xen, other commercial Type-1 hypervisors, such as those offered by VMware and Microsoft, have sufficiently similar structures that we believe the approach presented in this paper is applicable to them as well.

3.1 The Xen Platform

The Xen hypervisor relies on its control VM, Dom0, to provide a virtualized I/O path and host a system-wide registry and management toolstack.

Device Drivers. Xen delegates the control of PCI-based 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.

Virtualized devices are exposed to other VMs using a "split driver" model [17]. A backend driver, having direct control of the hardware, exposes virtualized devices to frontend drivers in the guest VMs. Frontend and backend drivers communicate over a shared memory ring, with the backend multiplexing requests from several frontends onto the underlying hardware. Xen is only involved in enforcing access control for the shared memory and passing synchronization signals. ACLs are stored in the form of *grant tables*, with permissions set by the owner of the memory.

Alternatively, Xen uses direct device assignment to allow VMs other than Dom0 to directly interface with passed-through hardware devices. Dom0 provides a virtual PCI bus, using a split driver, to proxy PCI configuration and interrupt assignment requests from the guest VM to the PCI bus controller. Device-specific operations are handled directly by the guest. Direct assignment can be used to move physical device drivers out of Dom0, in particular for PCI hardware that supports hardware-based IO virtualization (SR-IOV) [28].

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] instance, running either as a Dom0 process or in its own VM [44]. It has privileges to map any page of the guest's memory in order to emulate DMA operations.

XenStore. XenStore is a hierarchical key-value store that acts as a system-wide registry and naming service. It also provides a "watch" mechanism which notifies registered listeners of any modifications to particular keys in the store. Device drivers and the toolstack make use of this for inter-VM synchronization and device setup.

XenStore runs as a Dom0 process and communicates with other VMs via shared memory rings. Since it is required in the creation and boot-up of a VM, it relies on Dom0 privileges to access shared memory directly, rather than using grant tables.

Despite the simplicity of its interface with VMs, the complex, shared nature of XenStore makes it vulnerable to DoS attacks if a VM monopolizes its resources [14]. Because it is the central repository for configuration state in the system and virtually all components in the system depend on it, it is a critical component from a security perspective. Exploiting XenStore allows an attacker to deny service to the system as a whole and to perform most administrative operations, including starting and stopping VMs, and possibly abusing interfaces to gain access to guest memory or other guest VMs.

Other systems (including previous versions of Xen) have used a completely message-oriented approach, either as a point-to-point implementation or as a message bus. Having implemented all of these at various points in the past (and some of them more than once), our experience is that they are largely isomorphic with regard to complexity and decomposability.

Toolstack. The toolstack provides administrative functions for the management of VMs. It is responsible for creating, destroying, and managing the associated resources and privileges of VMs. Creating a VM requires Dom0 privileges to map guest memory, in order to load a kernel or virtual BIOS and to set up initial communication channels with XenStore and the virtual console. In addition, the toolstack registers newly created guests with XenStore.

System Boot. In a traditional Xen system, the boot process is simple: the hypervisor creates Dom0 during boot-up, which proceeds to initialize hardware and bring up devices and their associated backend drivers. XenStore is started before any guest VM is created.

pervisors, most of which assume the attacker has access to the host OS and compromises known OS vulnerabilities — for instance, using Windows exploits to compromise VMware Workstation. These attacks are not representative of our threat model and are excluded.

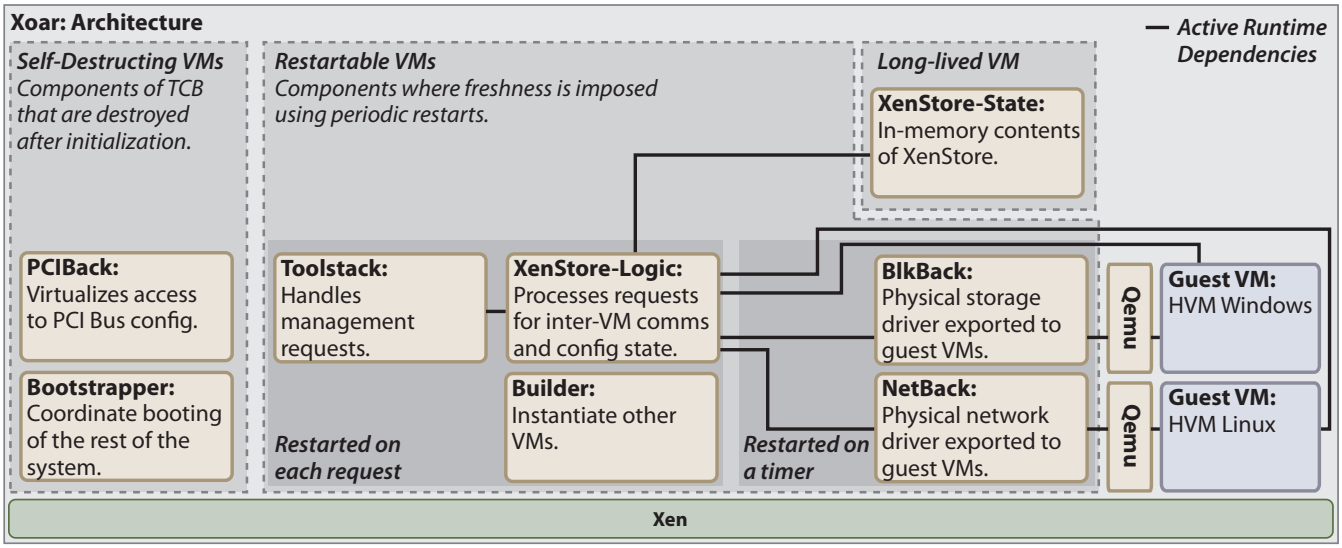

Figure 2: Architecture of Xoar. The figure above shows all the classes of service VMs along with the dependencies between them. For clarity, ephemeral dependencies (e.g., between the Builder and the VMs that it builds) are not shown. As suggested in the figure, a Qemu service VM is instantiated for the lifetime of each guest.

3.2 Xoar

Figure 2 shows the architecture of Xoar, and will be referred to throughout the remainder of this paper. In Xoar, the functionality of Xen's control VM has been disaggregated into nine classes of service VMs, each of which contains a single-purpose piece of control logic that has been removed from the original monolithic control VM. As is the case with the monolithic TCB, some components may have multiple instances, each serving different client VMs.

That these individual components may be instantiated more than once is important, as it allows them to be used as flexible building blocks in the deployment of a Xoar-based system. Figure 2 shows a single instance of each component other than the QemuVM. Later in the paper we will describe how multiple instances of these components, with differing resource and privilege assignments, can partition and otherwise harden the system as a whole.

From left to right, we begin with two start-of-day components that are closely tied to booting the hypervisor itself, *Bootstrapper* and *PCIBack*. These components bring up the physical platform and interrogate and configure hardware. In most cases this functionality is only required when booting the system and so these components are destroyed before any customer VMs are started. This is a useful property in that platform drivers and PCI discovery represent a large volume of complex code that can be removed prior to the system entering a state where it may be exposed to attacks.

While PCIBack is logically a start-of-day component, it is actually created after *XenStore* and *Builder*. XenStore is required to virtualize the PCI bus and the Builder is the only component capable of creating new VMs on the running system. PCIBack uses these components to create device driver VMs during PCI device enumeration by using `udev` [27] rules.

Three components are responsible for presenting platform hardware that is not directly virtualized by Xen. *BlkBack* and *NetBack* expose virtualized disk and network interfaces and control the specific PCI devices that have been assigned to them. For every guest VM running an unmodified OS, there is an associated *QemuVM* responsible for device emulation.

Once the platform is initialized, higher-level control facilities like the *Toolstacks* are created. The Toolstacks request the Builder to create guest VMs. As a control interface to the system, toolstacks are generally accessed over a private enterprise network, isolated from customer VM traffic.

As in Xen, a VM is described using a configuration file that is provided to the toolstack. This configuration provides runtime parameters such as memory and CPU allocations, and also device configurations to be provided to the VM. When a new VM is to be created, the toolstack parses this configuration file and writes the associated information into XenStore. Other components, such as driver VMs, have watches registered which are triggered by the build process, and configure connectivity between themselves and the new VM in response. While Xoar decomposes these components into isolated virtual machines, it leaves the interfaces between them unchanged; XenStore continues to be used to coordinate VM setup and tear down. The major difference is that privileges, both in terms of access to configuration state within XenStore and access to administrative operations in the hypervisor, are restricted to the specific service VMs that need them.

4. DESIGN

In developing Xoar, we set out to maintain functional parity with the original system and complete transparency with existing management and VM interfaces, including legacy support, without incurring noticeable overhead. This section discusses the approach that Xoar takes, and the properties that were considered in selecting the granularity and boundaries of isolation.

Our design is motivated by these three goals:

1. **Reduce privilege** Each component of the system should only have the privileges essential to its purpose; interfaces exposed by a component, both to dependent VMs and to the rest of the system, should be the minimal set necessary. This confines any successful attack to the limited capabilities and interfaces of the exploited component.

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

Figure 3: Privilege Assignment API

2. **Reduce sharing** Sharing of components should be avoided wherever it is reasonable; whenever a component is shared between multiple dependent VMs, this sharing should be made explicit. This enables reasoning and policy enforcement regarding the exposure to risk introduced by depending on a shared component. It also allows administrators to securely log and audit system configurations and to understand exposure after a compromise has been detected.

3. **Reduce staleness** A component should only run for as long as it needs to perform its task; it should be restored to a known good state as frequently as practicable. This confines any successful attack to the limited execution time of the exploited component and reduces the execution state space that must be tested and evaluated for correctness.

To achieve these goals, we introduce an augmented version of the virtual machine abstraction: the *service VM*. Service VMs are the units of isolation which host the service components of the control VM. They differ from conventional virtual machines in that only service VMs can receive any extra privilege from the hypervisor or provide services to other VMs. They are also the only components which can be shared in the system, aside from the hypervisor itself.

Service VMs are entire virtual machines, capable of hosting full OSes and application stacks. Individual components of the control VM, which are generally either driver or application code, can be moved in their entirety out of the monolithic TCB and into a service VM. The hypervisor naturally assigns privilege at the granularity of the tasks these components perform. As such, there is little benefit, and considerable complexity, involved in finer-grained partitioning.

Components receiving heightened privilege and providing shared services are targets identified by the threat model discussed in Section 2. By explicitly binding their capabilities to a VM, Xoar is able to directly harden the riskiest portions of the system and provide service-specific enhancements for security. The remainder of this section discusses the design of Xoar with regard to each of these three goals.

4.1 Privilege: Fracture the Monolithic TCB

A service VM is designated as such using a `serviceVM` block in a VM config file. This block indicates that the VM should be treated as an isolated component and contains parameters that describe its capabilities. Figure 3 shows the API for the assignment of the three privilege-related properties that can be configured: direct hardware assignment, privileged hypercalls, and the ability to delegate privileges to other VMs on creation.

Direct hardware assignment is already supported by many x86 hypervisors, including Xen. Given a PCI domain, bus, and slot number, the hypervisor validates that the device is available to be assigned and is not already committed to another VM, then allows the VM to control the device directly.

Hypercall permissions allow a service VM access to some of the privileged functionality provided by the hypervisor. The explicit white-listing of hypercalls beyond the default set available to guest

resource = [*provider*, *parameters*, ***constraint_group***=*tag*]

Figure 4: Constraint Tagging API

```
SELECT e1, e2 FROM log e1, log e2 WHERE
  e1.name = e2.name AND
  e1.action = 'create' AND
  e2.action = 'destroy' AND
  e1.dependency = 'NameOfCompromisedNetBack' AND
  overlaps(period_intersect(e1.time, e2.time),
           compromise_period);


SELECT e1.name FROM log e1 WHERE
  e1.dependency = 'NetBack' AND
  e1.dependency_version = vulnerable_version;
```

Figure 5: Temporal queries which search for guest VMs that depended on a service VM that was compromised (top) or vulnerable (bottom).

VMs allows for least-privilege configuration of individual service VMs. These permissions are translated directly into a Flask [41] policy, which is installed into the hypervisor.

Access to resources is restricted by delegating service VMs to only those Toolstacks allowed to utilize those resources to support newly created VMs. Attempts to use undelegated service VMs are blocked by the hypervisor, enabling coarse-grained partitioning of resources. In the private cloud example presented at the end of this section, each user is assigned a private Toolstack, with delegated service VMs, and has exclusive access to the underlying hardware.

4.2 Sharing: Manage Exposure

Isolating the collection of shared services in service VMs confines and restricts attacks and allows an explicit description of the relationships between components in the system. This provides a clear statement of configuration constraints to avoid exposure to risk and enables mechanisms to reason about the severity and consequences of compromises after they occur.

Configuration Constraints. A guest can provide constraints on the service VMs that it is willing to use. At present, a single constraint is allowed, as shown in Figure 4. The `constraint_group` parameter provides an optional user-specified tag and may be appended to any line specifying a shared service in the VM's configuration. Xoar ensures that no two VMs specifying different constraint groups ever share the same service VM.

Effectively, this constraint is a user-specified coloring that prevents sharing. By specifying a tag on all of the devices of their hosted VMs, users can insist that they be placed in configurations where they only share service VMs with guest VMs that they control.

Secure Audit. Xoar borrows techniques from past forensics systems such as Taser [18]. The coarse-grained isolation and explicit dependencies provided by service VMs makes these auditing approaches easier to apply. Whenever the platform performs a guest-related configuration change (e.g., the creation, deletion, pausing, or unpausing of a VM), Xoar logs the resulting dependencies to an off-host, append-only database over a secure channel. Currently, we use the temporal extension for Postgres.

Two simple examples show the benefit of this approach. First, the

Calls from within the service VM:
vm_snapshot ()
recoverybox_balloc (*size*)

VM configuration for restart policy:
restart_policy ([*(timer | event), parameters*])

Figure 6: Microreboot API

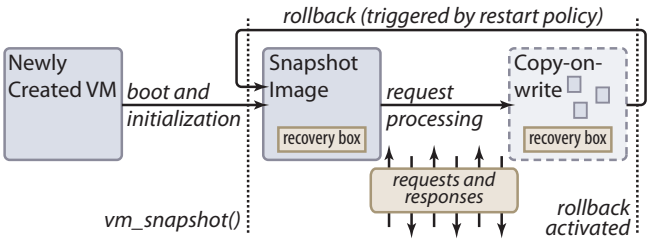

Figure 7: Rolling back to a known-good snapshot allows efficient microreboots of components.

top query in Figure 5 determines which customers could be affected by the compromise of a service VM by enumerating VMs that relied on that particular service VM at any point during the compromise. Second, providers frequently roll out new versions of OS kernels and in the event that a vulnerability is discovered in a specific release of a service VM after the fact, the audit log can be used to identify all guest VMs that were serviced by it.

4.3 Staleness: Protect VMs in Time

The final feature of service VMs is a facility to defend the *temporal* attack surface, preserving the freshness of execution state through the use of periodic restarts. This approach takes advantage of the observation from work on microreboots and "crash-only software" [10] that it is generally easier to reason about a program's correctness at the start of execution rather than over long periods of time.

Microreboots. Virtual machines naturally support a notion of rebooting that can be used to reset them to a known-good state. Further, many of the existing interfaces to control VM-based services already contain logic to reestablish connections, used when migrating a running VM from one physical host to another. There are two major challenges associated with microreboots. First, full system restarts are slow and significantly reduce performance, especially of components on a data path such as device drivers. Second, not all state associated with a service can be discarded since useful side-effects that have occurred during that execution will also be lost.

Snapshot and Rollback. Instead of fully restarting a component, it is snapshotted just after it has booted and been initialized, but before it has communicated with any other service or guest VM. The service VM is modified to explicitly snapshot itself at the time that it is ready to service requests (typically at the start of an event loop) using the API shown in Figure 6. Figure 7 illustrates the snapshot/rollback cycle. By snapshotting before any requests are served over offered interfaces, we ensure that the image is fresh. A complementary extension would be to measure and attest snapshot-based images, possibly even preparing them as part of a distribution and avoiding the boot process entirely.

We enable lightweight snapshots by using a hypervisor-based copy-on-write mechanism to trap and preserve any pages that are about to be modified. When rolling back, only these pages and the virtual CPU state need be restored, resulting in very fast restart times — in our implementation, between 4 and 25 ms, depending on the workload.

Restart Policy. While it is obvious when to take the snapshot of a component, it is less clear when that component should be rolled back. Intuitively, it should be as frequently as possible. However, even though rollbacks are quick, the more frequently a component is restarted, the less time it has available to offer a useful service. Xoar specifies rollback policy in the service VM's configuration file and we currently offer two policies: notification-based and timer-based. Restart policy is associated with the VM when it is instantiated and is tracked and enforced by the hypervisor.

In our notification-based policy, the hypervisor interposes on message notifications *leaving* the service VM as an indication that a request transaction has completed, triggering a restart. For low-frequency, synchronous communication channels (e.g., those that access XenStore), this method isolates individual transactions and resets the service to a fresh state at the end of every processed request. In other words, every single request is processed by a fresh version of the service VM.[2]

The overhead of imposing a restart on every request would be too high for higher-throughput, concurrent channels, such as NetBack and BlkBack. For these service VMs, the hypervisor provides a periodic restart timer that triggers restarts at a configurable frequency.

Maintaining State. Frequent restarts suffer from the exact symptom that they seek to avoid: the establishment of long-lived state. In rolling back a service VM, any state that it introduces is lost. This makes it particularly hard to build services that depend on keeping in-memory state, such as configuration registries, and services that need to track open connections.

We address this issue by providing service VMs with the ability to allocate a "recovery box" [3]. Originally proposed as a technique for high availability, this is a block of memory that persists across restarts. Service VM code is modified to store any long-lived state in one of these allocations and to check and restore from it immediately after a snapshot call. Memory allocated using this technique is exempted from copy-on-write.

Maintaining state across restarts presents an obvious attack vector — a malicious user can attempt to corrupt the state that is reloaded after every rollback to repeatedly trigger the exploit and compromise the system. To address this, the service treats the recovery box as an untrusted input and audits its contents after the rollback. Xen also tracks the memory pages in the allocation and forcibly marks all virtual addresses associated with them as non-executable.

Driver VMs, like NetBack and BlkBack, automatically renegotiate both device state and frontend connections in cases of failures or restarts, allowing them to discard all state at every restart. In these performance-critical components, however, any downtime significantly affects the throughput of guests. This downtime can be reduced by caching a very small amount of device and frontend state in a recovery box. The desired balance between security and performance can be chosen, as discussed in Section 7.2.

Components like XenStore, on the other hand, maintain a large amount of long-lived state for other components in the system. In

[2]This mechanism leaves open the possibility that an exploited service VM might not send the event that triggers the rollback. To cover this attack vector, the hypervisor maintains a watchdog timer for each notification-based service VM. If a timer goes off, the VM is rolled back; if the restart is triggered normally, the timer is reset.

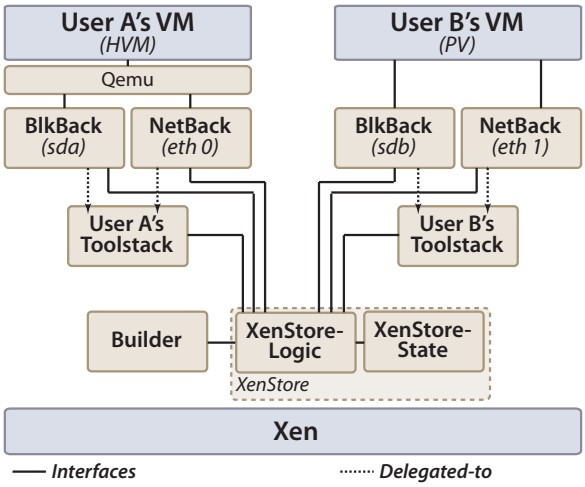

Figure 8: Partitioned configuration: In the configuration above, users A and B use isolated hardware and toolstacks and share interfaces only with XenStore and Xen itself.

such cases, this state can be removed from the service VM altogether and placed in a separate "state" VM that is accessible through a special-purpose interface. In Xoar, only XenStore, because of its central role in the correctness and security of the system, is refactored in this way (see Section 5.2). Only the processing and logic remain in the original service VM, making it amenable to rollbacks.

Per-request rollbacks force the attacker to inject exploit code into the state and have it triggered by another VM's interaction with XenStore. However, in the absence of further exploits, access control and guest ID authentication prevent the injection of such exploit code into sections of the state not owned by the attacking guest (see Section 5.2). Thus, an attack originating from a guest VM through XenStore requires an exploit of more than one service VM.

4.4 Deployment Scenarios

Public clouds, like Amazon Web Services, tightly pack many VMs on a single physical machine, controlled by a single toolstack. Partitioning the platform into service VMs, which can be judiciously restarted, limits the risks of sharing resources among potentially vulnerable and exposed VMs. Furthermore, dynamically restarting service VMs allows for in-place upgrades, reducing the window of exposure in the face of a newly discovered vulnerability. Finally, in the case of compromise, secure audit facilities allow administrators to reason, after the fact, about exposures that may have taken place.

Our design supports greater degrees of resource partitioning than this. Figure 8 shows a more conservative configuration, in which each user is assigned separate, dedicated hardware resources within the physical host and a personal collection of service VMs to manage them. Users manage their own service VMs and the device drivers using a private Toolstack with resource service VMs delegated solely to it.

5. IMPLEMENTATION

This section explains how the design described in Section 4 was implemented on the Xen platform. It begins with a brief discussion of how component boundaries were selected in fracturing the control VM and then describes implementation details and challenges faced during the development of Xoar.

5.1 Xoar Components

The division of service VMs in Xoar conforms to the design goals of Section 4; 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. Existing software and interfaces are reused to aid development and ease future maintenance. Table 1 augments Figure 2 by describing the classes of service VMs 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.

Virtualized devices mimic physical resources in an attempt to offer a familiar abstraction to guest VMs, making them ideal service VMs. Despite the lack of toolstack support, Xen has architectural support for driver VMs, reducing the development effort significantly. PCIBack virtualizes the physical PCI bus, while NetBack and BlkBack are driver VMs, exposing the required device backends for guest VMs. Further division, like separating device setup from the data path, yields no isolation benefits, since both components need to be shared simultaneously. This would also add a significant amount of IVC, conflicting with our design goals, and would require extensive modifications. Similarly, the serial controller is represented by a service VM that virtualizes the console for other VMs. Further details about virtualizing these hardware devices are discussed in Section 5.3 and Section 5.4.

Different aspects of the VM creation process require differing sets of privileges; placing them in the same service VM 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 the kernel, etc., and those that require access to XenStore to write entries necessary for the guest. Breaking this functionality apart along the lines of least privilege yields the Builder, a privileged service VM responsible for the hypervisor and guest memory operations, and the Toolstack, a service VM containing the management toolstack. While the Builder could be further 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. The Builder responds to build requests issued by the Toolstack via XenStore. Once building is complete, the Toolstack communicates with XenStore to perform the rest of the configuration and setup process.

5.2 XenStore

Our refactoring of XenStore is the most significant implementation change that was applied to any of the existing components in Xen (and took the largest amount of effort). We began by breaking XenStore into two independent service VMs: XenStore-Logic, which contains the transactional logic and connection management code, and XenStore-State, which contains the actual contents of the store. This division allows restarts to be applied to request-handling code on a per-request basis, ensuring that exploits are constrained in duration to a single request. XenStore-State is a simple key-value store and is the only long-lived VM in Xoar.

Unfortunately, partitioning and per-request restarts are insufficient to ensure the security of XenStore. As XenStore-Logic is responsible for enforcing access control based on permissions in the store itself, a compromise of that VM may allow for arbitrary accesses to the contents of the store. We addressed this problem with two techniques. First, access control checks are moved into a small monitor module in XenStore-State; a compromise of XenStore-Logic is now limited to valid changes according to existing permissions in the

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

Table 1: Components of Xoar. The "P" column indicates if the component is privileged. An "(R)" in the lifetime column indicates that the component can be restarted. Console is only mentioned for the sake of completeness. Since enterprise deployments typically disable console access, it is not part of the overall architecture.

store. Second, we establish the authenticity of accesses made by XenStore-Logic by having it declare the identity of the VM that it is about to service *before* reading the actual request. This approach effectively drops privilege to that of a single VM before exposing XenStore-Logic to any potentially malicious request, and makes the identity of the request made to XenStore-State unforgeable. The monitor refuses any request to change the current VM until the request has been completed, and an attempt to do so results in a restart of XenStore-Logic.

The monitor code could potentially be further disaggregated from XenStore-State and also restarted on a per-request basis. Our current implementation requires an attacker to compromise both XenStore-Logic and the monitor code in XenStore-State in succession, within the context of a single request, in order to make an unauthorized access to the store. Decoupling the monitor from XenStore-State would add limited extra benefit, for instance possibly easing static analysis of the two components, and still allow a successful attacker to make arbitrary changes in the event of the two successive compromises; therefore we have left the system is it stands.

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 requires a single component to multiplex all accesses to it. This space is used during device initialization, after which there is no further communication with PCIBack. We remove PCIBack from the TCB entirely after boot by destroying it, reducing the number of shared components in the system.

Hardware virtualization techniques like SR-IOV [28] allow the creation of virtualized devices, where the multiplexing is performed in hardware, obviating the need for driver VMs. However, provisioning new virtual devices on the fly requires a persistent service VM 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 VMs: NetBack and BlkBack

Driver VMs, like NetBack and BlkBack, use direct device assignment to directly access PCI peripherals like NICs and disk controllers, 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 virtualized backend driver. The Toolstack links a driver VM delegated to it to a guest VM by writing the appropriate frontend and backend XenStore entries during the creation of the guest, after which the guest and backend communicate directly using shared memory rings, without any further participation by XenStore.

Separating BlkBack from the Toolstack caused some problems as the existing management tools mount disk-based VM images as loopback devices with `blktap`, for use by the backend driver. After splitting BlkBack from the Toolstack, the disk images need to be created and mounted in BlkBack. Therefore, in Xoar, BlkBack runs a lightweight daemon that proxies requests from the Toolstack.

5.5 Efficient Microreboots

As described in Section 4.3, our snapshot mechanism copies memory pages which are dirtied as a service VM executes and restores the original contents of these pages during rollback, requiring a page allocation and deallocation and two copy operations for every dirtied page. Since many of the pages being modified are the same across several iterations, rather than deallocating the master copies of these pages after rollback, we retain them across runs, obviating the need for allocation, deallocation, and one copy operation when the same page is dirtied. However, this introduces a new problem: if a page is dirtied just once, its copy will reside in memory forever. This could result in memory being wasted storing copies of pages which are not actively required.

To address this concern, we introduced a "decay" value to the pages stored in the snapshot image. When a page is first dirtied after a rollback, its decay value is incremented by two, towards a maximum value. On rollback, each page's decay value is decremented. When this count reaches zero, the page is released.

5.6 Deprivileging Administrative Tools

XenStore and the Console require Dom0-like privileges to forcibly map shared memory, since they are required before the guest VM can set up its grant table mappings. To avoid this, Xoar's Builder creates grant table entries for this shared memory in each new VM, allowing these tools to use grant tables and function without any special privileges.

The Builder assigns VM management privileges to each Toolstack for the VMs that it requests to be built. A Toolstack can only manage these VMs, and an attempt to manage any others is blocked by the hypervisor. Similarly, it can only use service VMs that have been delegated to it. An attempt to use an undelegated service VM, for example a NetBack, for a new guest VM will fail. Restricting privileges this way allows for the creation of several Toolstack instances that run simultaneously. Different users, each with a private Toolstack, are able to partition their physical resources and manage their own VMs, while still guaranteeing strong isolation from VMs belonging to other users.

5.7 Developing with Minimal OSes

Bootstrapper and Builder are built on top of nanOS, a small, single-threaded, lightweight kernel explicitly designed to have the minimum functionality needed for VM creation. The small size and simplicity 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, demands more from its operating environment, and so is built on top of miniOS, a richer OS distributed with Xen.

Determining the correct size of OS to use is hard, with a fundamental tension between functionality and ease of use. Keeping nanOS so rigidly simple introduces a set of development challenges, especially in cases involving IVC. However, since these components have such high privilege, we felt that the improved security gained from reduced complexity is a worthwhile trade-off.

5.8 Implicit Assumptions about Dom0

The design of Xen does not mandate that all service components live in Dom0, however several components, including the hypervisor, implicitly hard-code the assumption that they do. A panoply of access control checks compare the values of domain IDs to the integer literal '0', the ID for Dom0. Many tools assume that they are running co-located with the driver backends and various paths in XenStore are hard-coded to be under Dom0's tree The toolstack expects to be able to manipulate the files that contain VM disk images, which is solved by proxying requests, as discussed in Section 5.4. The hypervisor assumes Dom0 has control of the hardware and configures signal delivery and MMIO and I/O-port privileges for access to the console and peripherals to Dom0. In Xoar, these need to be mapped to the correct VMs, with Console requiring the signals and I/O-port access for the console and PCIBack requiring the MMIO and remaining I/O-port privileges, along with access to the PCI bus.

6. SECURITY EVALUATION

Systems security is notoriously challenging to evaluate, and Xoar's proves no different. In an attempt to demonstrate the improvement to the state of security for commodity hypervisors, this section will consider a number of factors. First, we will evaluate the reduction in the size of the trusted computing base; this is an approach that we do not feel is particularly indicative of the security of a system, but has been used by a considerable amount of previous work and does provide some insight into the complexity of the system as a whole.

Permission	Bootstrapper	PCIBack	Builder	Toolstack	BlkBack	NetBack
Arbitrarily access memory	X		X			
Access and virtualize PCI devices		X				
Create VMs	X		X			
Manage VMs	X		X	X		
Manage assigned devices					X	X

Table 2: Functionality available to the service VMs in Xoar. Components with access to no privileged hypercalls are not shown. In Xen, Dom0 possesses all of these functionalities.

Component	Shared Interfaces
XenStore-Logic	XenStore-State, Console, Builder, PCIBack, NetBack, BlkBack, Guest
XenStore-State	XenStore-Logic
Console	XenStore-Logic
Builder	XenStore-Logic
PCIBack	XenStore-Logic, NetBack, BlkBack
NetBack	XenStore-Logic, PCIBack, Guest
BlkBack	XenStore-Logic, PCIBack, Guest
Toolstack	XenStore-Logic
Guest VM	XenStore-Logic, NetBack, BlkBack

Table 3: Interfaces shared between service VMs

Second, we consider how the attack surface presented by the control VM changes in terms of isolation, sharing, and per-component privilege in an effort to evaluate the exposure to risk in Xoar compared to other systems. Finally, we consider how well Xoar handles the existing published vulnerabilities first described in Section 2.

Much of this evaluation is necessarily qualitative: while we have taken efforts to evaluate against published vulnerabilities, virtualization on modern servers is still a sufficiently new technology with few disclosed vulnerabilities. Our sense is that these vulnerabilities may not be representative of the full range of potential attacks.

In evaluating Xoar's security, we attempt to characterize it from an attacker's perspective. One notable feature of Xoar is that in order for an adversary to violate our security claim, more than one service VM must have a vulnerability, and a successful exploit must be able to perform a stepping-stone attack. We will discuss why this is true, and characterize the nature of attacks that are still possible.

6.1 Reduced TCB

The Bootstrapper, PCIBack, and Builder service VMs are the most privileged components, with the ability to arbitrarily modify guest memory and control and assign the underlying hardware. These privileges necessarily make them part of the TCB, as a compromise of any one of these components would render the entire system vulnerable. Both Bootstrapper and PCIBack are destroyed after system initialization is complete, effectively leaving Builder as the only service VM in the TCB. As a result, the TCB is reduced from Linux's 7.6 million lines of code to Builder's 13,500 lines of code, both on top of the hypervisor's 280,000 lines of code.[3]

6.2 Attack Surface

Monolithic virtualization platforms like Xen execute service components in a single trust domain, with every component running at

[3]All lines of code were measured using David Wheeler's SLOCCount from `http://www.dwheeler.com/sloccount/`

Component	Arbitrary Code Execution	DoS	File System Access
Hypervisor	0 / 1	0 / 1	0 / 0
Device Emulation	8 / 8	3 / 3	3 / 3
Virtualized Drivers	1 / 1	1 / 1	0 / 0
XenStore	0 / 0	1 / 1	0 / 0
Toolstack	1 / 1	2 / 2	1 / 1

Table 4: Vulnerabilities mitigated in Xoar. The numbers represent total mitigated over total identified.

the highest privilege level. As a result, the security of the entire system is defined by that of the weakest component, and a compromise of any component gives an attacker full control of the system.

Disaggregating service components into their own VMs not only provides strong isolation boundaries, it also allows privileges to be assigned on a per-component basis, reducing the effect a compromised service VM has on the entire system. Table 2 shows the privileges granted to each service VM, which corresponds to the amount of access that an attacker would have on successfully exploiting it.

Attacks originating from guest VMs can exploit vulnerabilities in the interfaces to NetBack, BlkBack, or XenStore (see Table 3). An attacker breaking into a driver VM gains access only to the degree that other VMs trust that device. Exploiting NetBack might allow for intercepting another VM's network traffic, but not access to arbitrary regions of its memory. On hosts with enough hardware, resources can be partitioned so that no two guests share a driver VM.

Where components reuse the same code, a single vulnerability could be sufficient to compromise them all. Service VMs like NetBack, BlkBack, Console, and Toolstack run the same core Linux kernel, with specific driver modules loaded only in the relevant component. As a result, vulnerabilities in the exposed interfaces are local to the associated service VM, but vulnerabilities in the underlying framework and libraries may be present in multiple components. For better code diversity, service VMs could use a combination of Linux, FreeBSD, OpenSolaris, and other suitable OSes.

Highly privileged components like the Builder have very narrow interfaces and cannot be compromised without exploiting vulnerabilities in multiple components, at least one of which is XenStore. Along with the central role it plays in state maintenance and synchronization, this access to Builder makes XenStore an attractive target. Compromising XenStore-Logic may allow an attacking guest to store exploit code in XenStore-State, which, when restoring state after a restart, re-compromises XenStore-Logic. The monitoring code described in Section 5.2, however, prevents this malicious state from being restored when serving requests from any other guest VM, ensuring that they interact with a clean copy of XenStore.

6.3 Vulnerability Mitigation

With a majority of the disclosed vulnerabilities against Xen involving privilege escalation against components in Dom0, Xoar proves to be successful in containing all but two of them. Table 4 taxonomizes the vulnerabilities discussed in Section 2 based on the vulnerable component and the type of vulnerability, along with the number that are successfully mitigated in Xoar.

The 14 device emulation attacks are completely mitigated, as the device emulation service VM has no rights over any VM except the one the attacker came from. The two attacks on the virtualized device layer and the three attacks against the toolstack would only affect those VMs that shared the same BlkBack, NetBack, and Toolstack components. The vulnerability present in XenStore did not exist in our custom version. Since Xoar does not modify the hypervisor, the two hypervisor vulnerabilities remain equally exploitable.

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

Table 5: Memory requirements of individual components

One of the vulnerabilities in the virtualized drivers is against the block device interface and causes an infinite loop which results in a denial of service. Periodically restarting BlkBack forces the attacker to continuously recompromise the system. Since requests from different guests are serviced on every restart, the device would continue functioning with low bandwidth, until a patch could be applied to prevent further compromises.

7. PERFORMANCE EVALUATION

The performance of Xoar is evaluated against a stock Xen Dom0 in terms of memory overhead, I/O throughput, and overall system performance. Each service VM in Xoar runs with a single virtual CPU; in stock Xen Dom0 runs with 2 virtual CPUs, the configuration used in the commercial XenServer [12] 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 2.67 GHz Intel Xeon W3520 processor, 4 GB of RAM, a Tigon 3 Gigabit Ethernet card, and an Intel 82801JIR SATA controller with a Western Digital WD3200AAKS-75L9A0 320 GB 7200 RPM disk. VMX, EPT, and IOMMU virtualization are enabled. We use Xen 4.1.0 and Linux 2.6.31[4] pvops kernels for the tests. Identical guests running an Ubuntu 10.04 system, configured with two 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.

7.1 Memory Overhead

Table 5 shows the memory requirements of each of the components in Xoar. Systems with multiple network or disk controllers can have several instances of NetBack and BlkBack. Also, since users can select the service VMs to run, there is no single figure for total memory consumption. In commercial hosting solutions, console access is largely absent rendering the Console redundant. Similarly, PCIBack can be destroyed after boot. 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 750 MB Dom0 configuration used by XenServer. All performance tests compare a complete configuration of Xoar with a standard Dom0 Xen configuration.

7.2 I/O performance

Disk performance is tested using Postmark, with VMs' virtual disks backed by files on a local disk. Figure 9 shows the results of these tests with different configuration 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 eliminate performance artifacts due to disk performance). Figure 10 shows these results.

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

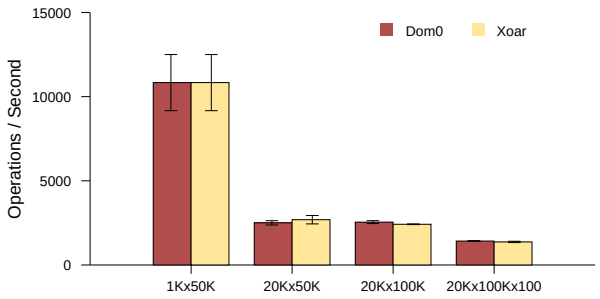

Figure 9: Disk performance using Postmark (higher is better). The x-axis denotes (files x transactions x subdirectories).

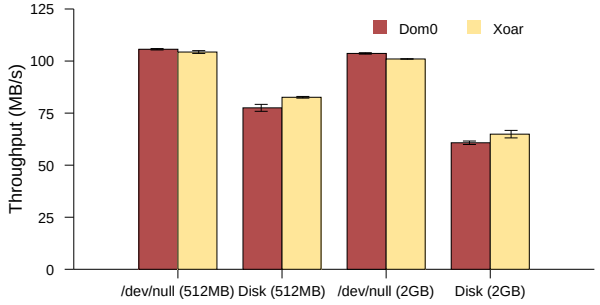

Figure 10: Network performance with `wget` (higher is better)

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 *increases* by 6.5%; we believe this is caused by the performance isolation of running the disk and network drivers in separate VMs.

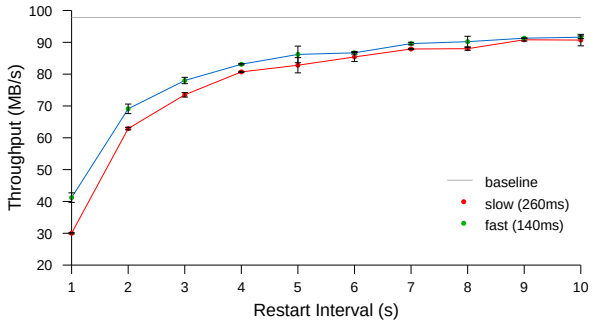

Figure 11: `wget` throughput while restarting NetBack at different time intervals

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

In the first (marked as "slow" in Figure 11), the device hardware state is left untouched during reboots; in the second ("fast"), some configuration data that would normally be renegotiated via Xen-

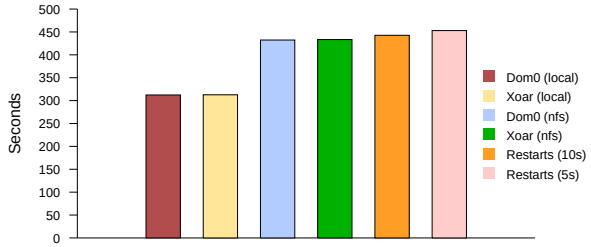

Figure 12: Linux kernel build run on Dom0 and Xoar, locally, over NFS and over NFS with NetBack restarts.

Store is persisted. In "slow" restarts the device downtime is around 260 ms, measuring from when the device is suspended to when it responds to network traffic again. The optimizations used in the "fast" restart reduce this downtime to around 140 ms.

Resetting every 10 seconds causes an 8% drop in throughput, as `wget`'s TCP connections respond to the breaks in connectivity. Reducing the interval to one second gives a 58% drop. Increasing it beyond 10 seconds 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.

7.3 Real-world Benchmarks

Figure 12 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%.

The *Apache Benchmark* is used to gauge the performance of an Apache web server serving a 10 KB static webpage 100,000 times to five simultaneous clients. Figure 13 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 8–9 ms, but with restarts, the values range from 3000 ms (at 5 and 10 seconds) to 7000 ms (at 1 second). As a result, the longest request interval is not shown in the figure.

Overall, the overhead of disaggregation is quite low. This is largely because driver VMs do not lengthen the data path between guests and the hardware: the guest VM communicates with NetBack or BlkBack, which drives the hardware. While the overhead of driver restarts is noticeable, as intermittent outages lead to TCP backoff, it can be tuned by the administrator to best match the desired combination of security and performance.

8. RELATED WORK

With the widespread use of VMs, the security of hypervisors has been studied extensively and several attempts have been made to address the problem of securing the TCB. This section looks at some of these techniques in the context of our functional requirements.

Build a Smaller Hypervisor. SecVisor [38] and BitVisor [40] are examples of tiny hypervisors, built with TCB size as a primary con-

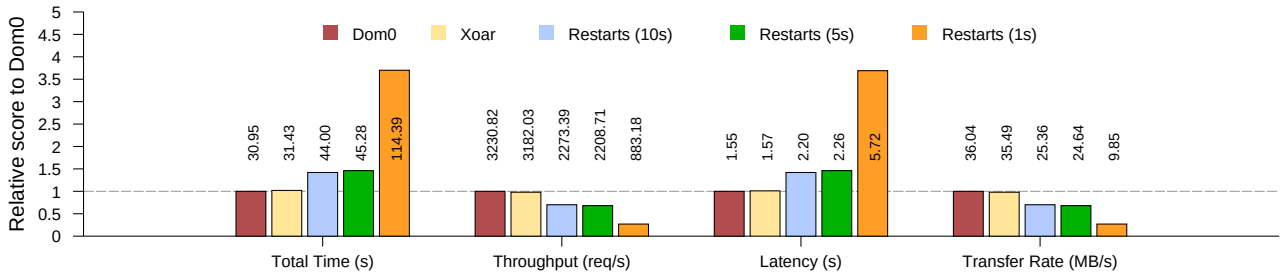

Figure 13: Apache Benchmark run on Dom0, Xoar, and Xoar with network driver restarts at 10s, 5s, and 1s.

cern, that use the interposition capabilities of hypervisors to retrofit security features for commodity OSes. While significantly reducing the TCB of the system, they do not share the multi-tenancy goals of commodity hypervisors and are unsuitable for such environments.

Microkernel-based architectures like KeyKOS [20] and EROS [39], its x86-based successor, are motivated similarly to Xoar and allow mutually untrusting users to securely share a system. Our Builder closely resembles the *factory* in KeyKOS. While multiple, isolated, independently administered UNIX instances, rather like VMs, can be hosted on EROS, this requires modifications to the environment and arbitrary OSes cannot be hosted. More recently, NOVA [42] uses a similar architecture and explicitly partitions the TCB into several user-level processes within the hypervisor. Although capable of running multiple unmodified OSes concurrently, the removal of the control VM and requirement for NOVA-specific drivers sacrifice hardware support for TCB size. Also, it is far from complete: it cannot run Windows guests and has limited toolstack support.

NoHype [23] 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 [13], memory sharing and compression [19, 32], and security enhancements [11, 30, 46, 16].

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 privileges of these components. Xen's XAPI toolstack is written in OCaml and benefits from the robustness that a statically typed, functional language provides [37]. Xen and Linux both have mechanisms to enforce fine-grained security policies [31, 36]. While useful, these techniques do not address the underlying concern about the size of the TCB.

Split Up the TCB, Reduce the Privilege of Each Part. Murray *et al.* [33] removed Dom0 userspace from the TCB 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 or remove the Dom0 kernel from the TCB, leaving the system vulnerable to attacks on exposed interfaces, such as network drivers.

Driver domains [17] allow device drivers to be hosted in dedicated VMs rather than Dom0, resulting in better driver isolation. QubesOS [35] uses driver domains in a single-user environment, but does not otherwise break up Dom0. Stub domains [44] isolate the Qemu device model for improved performance and isolation. Xoar builds on these ideas and extends them to cover the entire control VM.

9. DISCUSSION AND FUTURE WORK

This idea of partitioning a TCB is hardly new, with software partitioning having been explored in a variety of contexts before. Microkernels remain largely in the domain of embedded devices with relatively small and focused development teams (e.g., [26]), and while attempts at application-level partitioning have demonstrated benefits in terms of securing sensitive data, they have also demonstrated challenges in implementation and concerns about maintenance [7, 9, 24, 34], primarily due to the mutability of application interfaces.

While fracturing the largely independent, shared services that run in the control VM above the hypervisor, we observe that these concerns do not apply to nearly the same degree; typically the components are drivers or application code exposing their dominant interfaces either to hardware or to dependent guests. Isolating such services into their own VMs was a surprisingly natural fit.

While it is tempting to attribute this to a general property of virtualization, we also think that it was particularly applicable to the architecture of Xen. Although implemented as a monolithic TCB, several of the components were designed to support further compartmentalization, with clear, narrow communication interfaces.

We believe the same is applicable to Hyper-V, which has a similar architecture to Xen. In contrast, KVM [25] converts the Linux kernel itself into a hypervisor, with the entire toolstack hosted in a Qemu process. Due to the tight coupling, we believe that disaggregating KVM this aggressively would be extremely hard, more akin to converting Linux into a microkernel.

9.1 Lessons

In the early design of the system our overall rule was to take a practical approach to hardening the hypervisor. As usual, with the hindsight of having built the system, some more specific guidelines are clear. We present them here as "lessons" and hope that they may be applied earlier in the design process of future systems.

Don't break functionality. From the outset, the work described in this paper has been intended to be applied upstream to the open source Xen project. We believe that for VM security improvements to be deployed broadly, they must not sacrifice the set of functionality that has made these systems successful, and would not expect a warm reception for our work from the maintainers of the system if we were to propose that facilities such as CPU overcommit simply didn't make sense in our design.

This constraint places enormous limitations on what we are able to do in terms of hardening the system, but it also reduces the major argument against accepting new security enhancements.

Don't break maintainability. Just as the users of a virtualization platform will balk if enhancing security costs functionality, developers will push back on approaches to hardening a system that require additional effort from them. For this reason, our approach to hardening the hypervisor has been largely a *structural* one: individual service VMs already existed as independent applications in the monolithic control VM and so the large, initial portion of our work was simply to break each of these applications out into its own virtual machine. Source changes in this effort largely improved the existing source's readability and maintainability by removing hard-coded values and otherwise generalizing interfaces.

By initially breaking the existing components of the control VM out into their own virtual machines, we also made it much easier for new, alternate versions of these components to be written and maintained as drop-in replacements: our current implementation uses largely unchanged source for most of the service VM code, but then chooses to completely reimplement XenStore. The original version of XenStore still works in Xoar, but the new one can be dropped in to strengthen a critical, trusted component of the system.

There isn't always a single best interface. The isolation of components into service VMs was achieved through multiple implementations: some service VMs use a complete Linux install, some a stripped-down "miniOS" UNIX-like environment, and some the even smaller "nanOS", effectively a library for building small single-purpose VMs designed to be amenable to static analysis.

Preserving application state across microreboots has a similar diversity of implementation: driver VMs take advantage of a recovery-box-like API, while for the reimplementation of XenStore it became more sensible to split the component into two VMs, effectively building our own long-lived recovery box component.

Our experience in building the system is that while we might have built simpler and more elegant versions of each of the individual components, we probably couldn't have used fewer of them without making the system more difficult to maintain.

9.2 Future Work

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 a minor modification of XenStore keys. Unfortunately, these are not applicable to long-lived components with state like XenStore and the hypervisor itself. XenStore could potentially be restarted by persisting its state to disk. Restarting Xen under executing VMs, however, is more challenging. We would like to explore techniques like those in ReHype [29], but using *controlled* reboots to safely replace Xen, allowing the complete virtualization platform to be upgraded and restarted without disturbing the hosted VMs.

Although the overall design allows for it, our current implementation does not include cross-host migration of VMs. We are in the process of implementing a new service VM that contains the live VM migration toolset to transmit VMs over the network. While this component is not currently complete, it has begun to demonstrate an additional benefit of disaggregation: the new implementation strikes a balance between the implementation of a feature that requires considerable privilege to map and monitor changes to a VM's memory in the control VM, and the proposal to completely internalize migration within the guest itself [13]. Xoar's live migration tool allows the guest to delegate access to map and monitor changes to its memory to a trusted VM, and allows that VM to run, much like the QemuVM, for as long as is necessary. We believe that this technique will further apply to other proposals for interposition-based services, such as memory sharing, compression, and virus scanning.

10. CONCLUSION

Advances in virtualization have spurred demand for highly-utilized, low-cost centralized hosting of systems in the cloud. The virtualization layer, while designed to be small and secure, has grown out of a need to support features desired by enterprises.

Xoar is an architectural change to the virtualization platform that looks at retrofitting microkernel-like isolation properties to the Xen hypervisor without sacrificing any existing functionality. It divides the control VM into a set of least-privilege service VMs, which not only makes any sharing dependencies between components explicit, but also allows microreboots to reduce the temporal attack surface of components in the system. We have achieved a significant reduction in the size of the TCB, and address a substantial percentage of the known classes of attacks against Xen, while maintaining feature parity and incurring very little performance overhead.

11. ACKNOWLEDGMENTS

We would like to thank our shepherd, Bryan Ford, the anonymous reviewers, Steve Hand, Derek Murray, Steve Gribble, Keir Fraser, David Lie, and the members of the systems research groups at the University of British Columbia and at the University of Cambridge for their suggestions and feedback. This work was partially supported through funding from the NSERC Internetworked Systems Security Network (ISSNet) and from the Communications Security Establishment Canada (CSEC).

12. REFERENCES

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

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM SOSP*, pages 1–14, Oct. 2009.

[3] M. Baker and M. 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.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM SOSP*, pages 164–177, Oct. 2003.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, December 2007.

[6] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC*, pages 41–46, Apr. 2005.

[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proc. 5th USENIX NSDI*, pages 309–322, Apr. 2008.

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

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

[10] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proc. 6th USENIX OSDI*, pages 31–44, Dec. 2004.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. 13th ASPLOS*, pages 2–13, Mar. 2008.

[12] Citrix Systems, Inc. *Citrix XenServer 5.6 Admininistrator's Guide*. June 2010.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. 2nd USENIX NSDI*, pages 273–286, May 2005.

[14] P. Colp. [xen-devel] [announce] xen ocaml tools. `http://lists.xensource.com/archives/html/xen-devel/2009-02/msg00229.html`.

[15] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. 5th USENIX NSDI*, pages 161–174, Apr. 2008.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. 15th ACM CCS*, pages 51–62, Oct. 2008.

[17] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st OASIS*, Oct. 2004.

[18] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser intrusion recovery system. In *Proc. 20th ACM SOSP*, pages 163–176, Oct. 2005.

[19] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *Proc. 8th Usenix OSDI*, pages 85–93, Oct. 2008.

[20] N. Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[21] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proc. 11th ACM SIGOPS EW*, Sept. 2004.

[22] K. Kappel, A. Velte, and T. Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, 1st edition, 2010.

[23] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proc. 37th ACM ISCA*, pages 350–361, June 2010.

[24] D. Kilpatrick. Privman: A library for partitioning applications. In *Proc. USENIX ATC*, pages 273–284, June 2003.

[25] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. `kvm`: the Linux virtual machine monitor. In *Proc. Linux Symposium*, pages 225–230, July 2007.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. 22nd ACM SOSP*, pages 207–220, Oct. 2009.

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

[28] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. Application note 321211-002, Intel Corporation, Jan. 2011.

[29] M. Le and Y. Tamir. ReHype: Enabling VM survival across hypervisor failures. In *Proc. 7th ACM VEE*, pages 63–74, Mar. 2011.

[30] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proc. 17th USENIX Security Symposium*, pages 243–258, July 2008.

[31] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. USENIX ATC*, pages 29–42, June 2001.

[32] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proc. USENIX ATC*, pages 1–14, June 2009.

[33] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proc. 4th ACM VEE*, pages 151–160, Mar. 2008.

[34] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proc. 12th USENIX Security Symposium*, pages 231–242, Aug. 2003.

[35] J. Rutkowska and R. Wojtczuk. *Qubes OS Architecture*. Version 0.3. Jan. 2010. `http://qubes-os.org/`.

[36] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Proc. 21st ACSAC*, pages 276–285, Dec. 2005.

[37] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using functional frogramming within an industrial product group: perspectives and perceptions. In *Proc. 15th ICFP*, pages 87–92, Sept. 2010.

[38] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. 21st ACM SOSP*, pages 335–350, Oct. 2007.

[39] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. 17th ACM SOSP*, pages 170–185, Dec. 1999.

[40] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *Proc. 5th ACM VEE*, pages 121–130, Mar. 2009.

[41] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. 8th USENIX Security Symposium*, pages 123–139, Aug. 1999.

[42] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. 5th EuroSys*, pages 209–222, Apr. 2010.

[43] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39(5):44–51, May 2006.

[44] S. Thibault and T. Deegan. Improving performance by embedding HPC applications in lightweight Xen domains. In *Proc. 2nd HPCVIRT*, Mar. 2008.

[45] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proc. ACM SIGMOD*, pages 231–242, June 2010.

[46] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. 16th ACM CCS*, pages 545–554, Nov. 2009.

[47] J. Wilkes, J. Mogul, and J. Suermondt. Utilification. In *Proc. 11th ACM SIGOPS EW*, Sept. 2004.