Virtual Switching Without a Hypervisor for a More Secure Cloud

Xin Jin
Princeton University

Eric Keller
University of Pennsylvania

Jennifer Rexford
Princeton University

Abstract

Cloud computing leverages virtualization to offer resources on demand to multiple "tenants". However, sharing the server and network infrastructure creates new vulnerabilities, where one tenant can attack another by compromising the underlying hypervisor. We design a system that supports virtualized networking using software switches *without a hypervisor*. In our architecture, the software switch runs in a Switch Domain (DomS) that is separate from the control VM. Both the guest VMs and DomS run directly on the server hardware, with processing and memory resources allocated in advance. Each guest VM interacts with the software switch through a shared memory region using periodic polling to detect network packets. The communication does not involve the hypervisor or the control VM. In addition, any software bugs that crash the software switch do not crash the rest of the system, and a crashed switch can be easily rebooted. Experiments with our initial prototype, built using Xen and Open vSwitch, show that the combination of shared pages and polling offers reasonable performance compared to conventional hypervisor-based solutions.

1 Introduction

Cloud computing leverages virtualization to offer computing, storage, and networking resources on demand to multiple "tenants". Server virtualization enables multiple virtual machines (VMs) to run on the same host, and share a connection to the network. To virtualize the underlying network, cloud providers increasingly rely on software switches that run on the hypervisor or a control VM (e.g., Dom0 in Xen) on the server. These switches provide access control, resource and namespace isolation between tenants, efficient communication between VMs on the same server, and support for live VM migration. Popular examples include Open vSwitch [1], VMware's vSwitch [2], and Cisco's Nexus 1000V Switch [3].

Despite the many benefits of cloud computing, sharing the server and network infrastructure creates new security vulnerabilities that make some companies reluctant to move their applications and data to the cloud [4]. The virtualization layer (including the software switch) is quite complex, forming a large trusted computing base. For example, there are $\sim$200K lines of code in the Xen hypervisor, $\sim$600K in the emulator, and more than 1M in the host operating system. With such a large amount of code, bugs in the hypervisor are inevitable, as evinced by the many bug reports for Xen in NIST's National Vulnerability Database [5]. Malicious tenants can exploit these vulnerabilities to attack software switches and the virtualization layer and gain control of the physical server. The attacker can then obstruct or access other VMs, compromising confidentiality, integrity, and availability of other tenants' code and data.

Recent research proposes ways to reduce the attack surface by dividing the virtualization layer into smaller, independent components [6] or removing the hypervisor entirely [7, 8]. In NoHype architecture [7, 8], guest VMs run directly on the server hardware—without an underlying hypervisor. As part of booting the VM, NoHype allocates processor cores, physical memory pages, and virtual network interface cards (NICs) to the guest VM, and performs all necessary system discovery. This obviates the need for guest VMs to perform "VM exits" to access services normally provided by a hypervisor. While a promising way to improve security in the cloud, removing the hypervisor makes it difficult to support software switches. A software switch could be supported by bouncing packets off the NIC or a hardware switch, but this would consume excessive resources on the PCI bus and the access link.

In this paper, we design a system that allows virtual machines to communicate using software switches, without relying on an underlying hypervisor. Rather than relaying interrupts through a hypervisor, each VM interacts with the software switch through a shared memory region using periodic polling to detect network packets. Both the guest VM and the software switch run directly on the server hardware, with processing and memory resources allocated in advance. Each VM has its own dedicated region of memory for communicating with the software switch, allowing permissions in the hardware page tables to block access by other VMs. Since the software switch does not run in a hypervisor (or in Dom0), any software bugs that lead to the switch being compromised do not open the potential to compromise the rest of the system, and the switch can be easily "microrebooted" as in earlier work [6] on Dom0 disaggregation. Experiments with our initial prototype, built using Xen and Open vSwitch, show that communicating through shared pages offers reasonable performance compared to conventional hypervisor-based solutions (limited only by optimizations of kernel to user space communication that we have yet to incorporate).

The paper is organized as follows. We first provide background information in Section 2 on related technology and trends that inspire aspects of our architecture. We then present our architecture in Section 3. The details of our prototype system are described in Section 4. We then evaluate our prototype in Section 5. We conclude and discuss future work in Section 6.

2 Background: Virtualization Layer

Before we describe the design of our system, we present some important background material on server virtualization and networking support. First, we briefly describe how Xen performs networking in the control VM (i.e., Dom0). Other virtualization technology uses similar mechanisms, but we will mainly focus on Xen in this paper. Then, we explain how Dom0 disaggregation can improve security and reliability by dividing the control VM's functionality into several smaller components. Finally, we discuss how NoHype goes one step further by enabling guest VMs to run directly on the underlying hardware, without a hypervisor.

2.1 Xen Networking in Dom0

By default, Xen provides network functionality in the control VM, which has direct access to the physical NICs, as shown in Figure 1. For each guest VM, Xen creates a pair of connected virtual Ethernet interfaces—one in Dom0, and the other in the corresponding guest VM. For example, *vif1.0* in Dom0 connects to *eth0* of Guest VM 1. These virtual interfaces can be realized through para-virtualized drivers, as shown, or device emulation. Para-virtualized drivers use a split driver model, with a frontend network driver (Netfront) installed in the guest VM to provide *eth0*, and a backend network driver

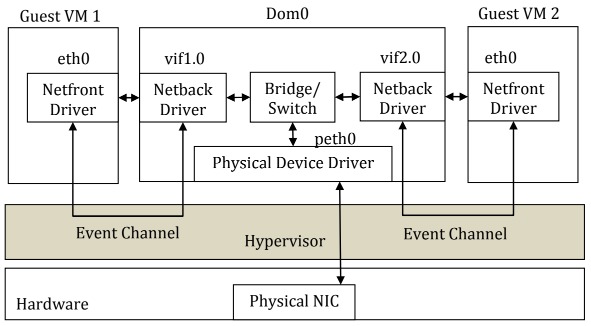

Figure 1: Xen Networking. For each guest VM, a pair of connected virtual Ethernet interfaces is created. They communicate with the help of hypervisor.

(Netback) installed in Dom0 to provide *vif1.0*. Netfront and Netback use a Xen event channel to send interrupts and use shared memory to transmit network packets. For hardware virtual machines (HVMs) that run an unmodified OS in guest VMs, Dom0 emulates virtualized network devices for guest VMs. By default, Dom0 creates a Linux bridge that connects all of these virtual Ethernet devices, so that it can multiplex all guest VMs' network traffic onto the underlying hardware. Instead of simply bridging, it is more attractive for infrastructure providers to connect the virtual interfaces to a software switch, which can provide more functionality.

2.2 Dom0 Disaggregation

This default architecture with everything, including the software switch, drivers, and device emulation, running in Dom0, is not secure. As such, researchers have worked toward disaggregating this functionality into smaller, single-purpose and mostly independent components that run in separate VMs.

The idea originates from the concept of driver domains in Xen 2.0, where a backend driver can run in a separate domain and multiplexes requests from guest VMs' frontends [9]. Then in Xen 3.3, the concept of stub domain is introduced which extends the idea of driver domains to other control VM's components–namely the device emulation.

More recently, Xoar [6] takes the idea of Dom0 disaggregation further. It breaks Dom0 into single-purpose components called service VMs and introduces the ability to restart components, which they call microreboot. Microrebooting components to known-good snapshots can reduce the temporal attack surface of individual components and allow developers to reason about specific software state. Sharing of service VMs by guests is also configurable and auditable, making the whole system more secure.

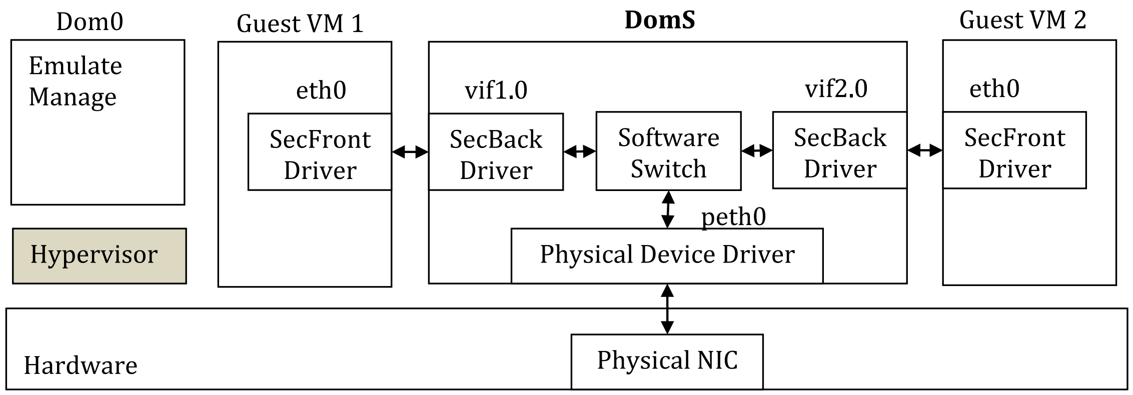

Figure 2: Secure Software Switch Architecture. A separate domain, DomS, runs a software switch and handles all network traffic. SecFront and SecEnd create a pair of connected virtualized Ethernet interfaces. The communication between DomS and a guest VM does not involve the hypervisor or Dom0.

2.3 Disengaging the Hypervisor

With the idea of Dom0 disaggregation, these buggy components are isolated from one another. However, for a secure system in a multi-tenant environment, their reliance on hypervisor involvement is another critical source of concern. The large code base of today's hypervisors might contain bugs that can be utilized by malicious customers, and thus be a threat to the whole system.

NoHype is an architecture that eliminates the hypervisor attack surface by enabling the guest VMs to run natively on the underlying hardware while maintaining the ability to run multiple VMs concurrently [8] [7]. In NoHype, the guest VM does not interact with the hypervisor or the control VM (Dom0) at all. NoHype pre-allocates processor cores and memory resources for each guest VM and performs all system discovery during bootup. Guest VMs on NoHype only use virtualized I/O devices and directly contact with the underlying hardware. Given the need for using only virtualized I/O devices in NoHype, software switching is only possible by bouncing packets through the network device over the PCI bus. Then the communication between two co-located VMs would traverse the PCI bus four times. This is not efficient, nor is it scalable.

3 Architecture

As discussed above, there are two main security threats to the communication between the software switch and guest VMs: their communication requires hypervisor involvement; the software switch is coupled with the control VM (Dom0). We propose an architecture here which removes these two threats for a more secure cloud, as shown in Figure 2. Our architecture removes the first threat by confining the communication channel between the software switch and a guest VM to a piece of shared memory. Both only use polling on the shared memory region to check for incoming packets, so that they do not need hypervisor involvement. Our architecture removes the second threat by putting the software switch in a separate domain, which we call Switch Domain (DomS). DomS exchanges all necessary information with the hypervisor and Dom0 during initialization, and then runs without the hypervisor and Dom0 involvement. Therefore any compromise of DomS would only produce very limited damage to the whole system.

3.1 Eliminating the Hypervisor-Guest Interaction

In virtualization today, the communication between the software switch and guest VMs requires hypervisor involvement. The software switch and guest VMs rely on the hypervisor to send interrupts to coordinate their operations. However, the hypervisor is a potential attack surface which can be utilized by malicious customers to compromise the whole system. In order to remove such a threat, we confine the communication between a guest VM and software switch to a piece of shared memory. So, the communication channel is minimal as it is composed of only some buffer pages with well-defined packet formats. Furthermore, both sides use polling to check for incoming packets and do not interact with the hypervisor.

- **Shared Memory:** Each guest VM uses a shared memory region, which is created during initialization, to communicate with the software switch. There are two first-in-first-out (FIFO) buffers in this shared memory region, each of which is used for communication in one direction. For each FIFO buffer, one VM writes network packets on one side and updates the write pointer; the other VM reads network packets on the other side and updates the read pointer. Since the write pointer is only modified by one VM and the read pointer is only modified by the other VM, the modifications are lock-free and do not need additional mechanisms to synchronize or coordinate.

- **Polling Only:** Both sides, guest VMs and the software switch, use polling to detect network packets. They continuously check the FIFO buffers for incoming packets. They do not rely on any hypervisor or Dom0 support. Recall that in native Xen, when one VM sends out a packet using a frontend driver, it uses the Xen event channel to inform the backend driver in Dom0 (or driver domain) of this new packet. Compared to that, we eliminate all these interactions in our architecture.

3.2 Limiting Damage From a Compromised Switch

Another threat with today's system is that the software switch is coupled with the control VM (Dom0). Such coupling can cause serious security problems to the system. A malicious guest VM could conceivably compromise the software switch through the shared memory region, and then control or crash the whole system.

In order to remove this threat, we introduce a *Switch Domain (DomS)* which runs the software switch, as shown in Figure 2. In this way, we further decouple the software switch from the control VM. DomS has the same privilege as a guest VM. In the runtime, there is no interaction with Dom0. So even if the DomS is completely compromised, it does not affect Dom0. Dom0 can easily create a new VM to serve as a switch domain. Guest VMs only lose network connection for a short time, and can gain network connection again after the new switch domain is created. Compared to native Xen, our approach greatly improves the availability of the whole system.

The switch domain here follows the concept of Dom0 disaggregation, though doing it for network functionality has largely been ignored before. The switch domain handles all VM network traffic. We should notice that the motivation of the switch domain we propose here is different from that of a driver domain. The purpose of the switch domain is to isolate the software switch, while the purpose of the driver domain is to isolate the physical device driver.

We also leverage the idea of NoHype to further isolate DomS from the hypervisor. The function of the hypervisor in our architecture is only to pre-allocate resources for booting VMs and clean the environments after shutting down VMs. The challenge here is that after initialization DomS cannot interact with the hypervisor to set up shared memory with newly-created guest VMs. After initialization, we don't want it to interact with the hypervisor any more because that would introduce an avenue for an attacker that has compromised the switch software to attack the system software (Dom0 and the hypervisor). If a new guest VM is created, it is difficult to create a shared memory region between this newly-created VM and DomS and then inform DomS of this information without hypervisor involvement.

The approach we use here is to pre-allocate all pages when booting DomS. These pages can be regarded as a pool. A FIFO buffer consists of several pages in the pool. When booting up a new guest VM, it just requests two FIFO buffers from the pool and maps them into its own address space. Then it can use these buffers to communicate with DomS. In this way, DomS does not need to interact with the hypervisor after it has been set up.

4 Prototype

Our prototype is based on Xen 4.1, Linux 3.1 and Open vSwitch 1.3. We make use of Xen to boot VMs. But after that DomS and guest VMs do not interact with the hypervisor any more. Incorporating the prototype to NoHype is part of our future work.

We use the split driver model in our prototype. We implement two drivers, the SecFront driver installed by the guest VM, and the SecBack driver installed by DomS. For each VM, we use SecFront and SecBack to create a pair of connected virtual Ethernet interfaces. The interfaces in DomS are connected to the software switch, as shown in Figure 2. These drivers are kernel modules and are loaded when booting up a VM. In order to further describe the details of how our prototype works, we explain it step by step.

4.1 SecBack in Switch Domain

DomS is created before any guest VMs are created. In our prototype, we use the function of Xen to create a DomS. During its initialization, SecBack is loaded and interacts with the hypervisor. SecBack allocates several pairs of FIFO buffers which serves as a memory pool. Each guest VM can request one pair of FIFO buffers, each of which is used for communication in one direction. The total size of this pool is configurable, and is fixed after bootup. Since the maximum number of guest VMs running on the same physical machine is limited, the pool does not consume too much memory. SecBack uses Xen's grant table mechanism to create a shared memory region between the DomS and each guest VM. After creating the memory pool, SecBack uses hypercalls to notify the hypervisor of this information. Then DomS runs on its own and does not interact with the hypervisor or Dom0 in the runtime.

4.2 SecFront in Guest VM

After initializing DomS, the system can receive requests from customers and dynamically create and destroy guest VMs. Again, we make use of Xen to set up guest VMs. During the initialization of a guest VM, the frontend driver, SecFront, is loaded by the guest VM's operating system. SecFront makes a request to the hypervisor for a pair of FIFO buffers in DomS. Since the FIFO buffers have all been pre-allocated by DomS, the interaction here is only between the guest VM and the hypervisor, and does not involve DomS. SecFront also makes use of the grant table mechanism in Xen to get access to the FIFO buffers and then maps them into its own address space. After that the guest VM can use the FIFO buffers to communicate with DomS. After all these buffers have been set up, the guest VM can use these buffers to communicate and does not need to interact with the hypervisor or Dom0 in the runtime.

4.3 Network Communication

For each guest VM, the SecFront driver handles all network traffic. The SecFront driver creates a virtualized Ethernet interface in the guest VM. A network packet in a guest VM is first sent to the SecEnd driver by the SecFront driver. The SecEnd driver also creates a virtualized Ethernet interface in DomS, and that interface is connected to the software swtich. The packet received by the SecEnd driver is then handled by the software switch in DomS. If the destination is on the same physical machine, the software switch sends the packet to that VM by the SecEnd driver. If the destination is on another physical machine, the software switch sends it out using physical NICs (or alternatively, to a driver domain when then sends it out). The transmission between a SecFront driver and a SecEnd driver is through the shared FIFO buffers. One driver writes one packet to the buffer, and the other driver uses polling to detect and receive it. In this way, their communication does not involve the hypervisor or Dom0.

5 Evaluation

In this section, we present the evaluation of our prototype. The test server has an Intel XEON W5580 processor with 8 cores and 6GB of RAM. We first analyze the impact of FIFO size on network performance. Then we analyze the impact of polling period on network performance. Finally, we compare the network performance of our system with native Xen.

In the evaluation, we create a DomS and a guest VM on the test server, each of which is configured with 1 core and 1 GB of RAM. We use netperf [10] to measure the performance of our system.

5.1 Impact of FIFO Size

In this group of experiments, we evaluate the impact of FIFO size on network performance. We fix the polling period at 1ms, and vary the number of pages of each FIFO buffer. A page is 4KB. We use netperf to measure the throughput between DomS and the guest VM. The message size is 1KB. The results are shown in Figure 3.

From the figure, we can see that the throughput increases along with the FIFO size. With only 256 pages (or 1MB of memory), the system effectively reaches the maximum throughput. This means that even for very high network performance, the system does not consume much memory for communication between DomS and a guest VM.

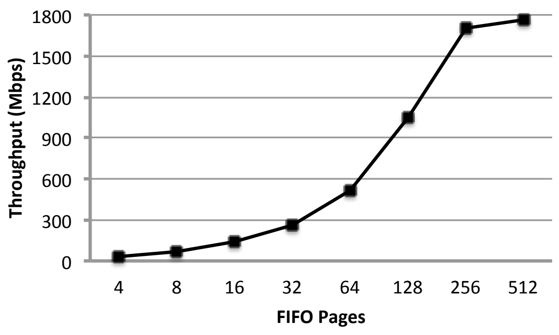

Figure 3: FIFO Size vs. Throughput

5.2 Impact of Polling Period

In this group of experiments, we evaluate the impact of polling period on network performance. We fix the number of FIFO pages at 256, and vary the polling period. We also use netperf to measure the throughput between DomS and the guest VM. The message size is 1KB. The results are shown in Figure 4.

The figure shows that the throughput of the system increases when the polling period decreases. This is reasonable since when the system checks the shared memory region for coming packets more frequently, it can receive more packets at same time. But if the system polls more frequently, it will consume more CPU resource.

Further exploration in the tradeoff between performance and CPU overhead is part of our future work.

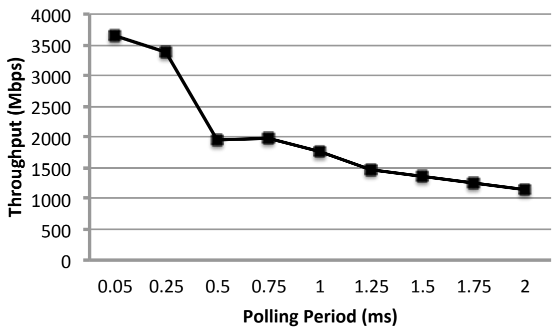

Figure 4: Polling Period vs. Throughput

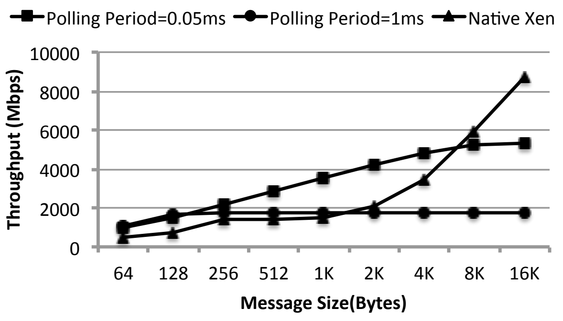

Figure 5: Comparison With Native Xen

5.3 Comparison With Native Xen

Now we compare the networking performance of our system with that of native Xen. In our system, we still evaluate the throughput between DomS and a guest VM, each of which is configured with 1 core and 1 GB of RAM. We set the FIFO pages to 256. We set the polling period to 0.05ms and 1ms. In native Xen, we create a guest VM, and evaluate the throughput between Dom0 and the guest VM. The guest VM is configured with 1 core and 1 GB of RAM. We use different message sizes to compare the performance of the two systems. The results are shown in Figure 5.

From the figure, we can see that our system has higher throughput than native Xen when the message size is between 64B and 4KB. This is because we simply use dedicated memory for network communication and VMs only use polling to check new packets, while in native Xen, there are many transitions among guest VMs, the hypervisor, and Dom0, which degrade the performance. But our system does not outperform Xen when the message size is bigger than 8KB. This is because Xen does some optimization for transmission. It enables "zero-copy" transmission of network data directly from user-space buffers. This "zero-copy" optimization greatly reduces overhead for large messages sent in user space. Our prototype does not have that feature now, though we intend to incorporate it as part of future work. However, even without that feature, our system still achieves good performance, and performs even better than Xen on small messages.

6 Conclusion and Future Work

In this work we improved security in a multi-tenant environment by enabling the use of a software switch in an environment without hypervisor involvement. However, while the attack surface is small, there is an attack surface. While we eliminated the opportunity for damage to the rest of the system due to a compromised software switch, the software switch is still an important component that guest VMs rely on. As such, for future work, we intend to further improve the security of the system with techniques for detecting when the switch has been compromised and remediation of the vulnerability.

References

[1] openvswitch.org, "Open vSwitch." `http://openvswitch.org/`, December 2011.

[2] VMware, "VMware vSphere Distributed Switch." `http://www.vmware.com/products/vnetwork-distributed-switch/overview.html`, December 2011.

[3] C. Systems, "Cisco Nexus 1000V Series Switches." `http://www.cisco.com/en/US/products/ps9902/index.html`, December 2011.

[4] F. Gens, "IT cloud services user survey, pt.2: Top benefits & challenges," Oct. 2008. `http://blogs.idc.com/ie/?p=210`.

[5] "CVE-2007-4993." `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4993`.

[6] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *ACM Symposium on Operating Systems Principles*, pp. 189–202, 2011.

[7] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *ACM Conference on Computer and Communications Security*, pp. 401–412, 2011.

[8] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *International Symposium on Computer Architecture*, pp. 350–361, 2010.

[9] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," in *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[10] Netperf. `http://www.netperf.org/`, December 2011.