Eliminating the Hypervisor Attack Surface for a More Secure Cloud

Jakub Szefer, Eric Keller, Ruby B. Lee and Jennifer Rexford
Princeton University
{szefer, ekeller, rblee, jrex}@princeton.edu

ABSTRACT

Cloud computing is quickly becoming the platform of choice for many web services. Virtualization is the key underlying technology enabling cloud providers to host services for a large number of customers. Unfortunately, virtualization software is large, complex, and has a considerable attack surface. As such, it is prone to bugs and vulnerabilities that a malicious virtual machine (VM) can exploit to attack or obstruct other VMs — a major concern for organizations wishing to move "to the cloud." In contrast to previous work on hardening or minimizing the virtualization software, we eliminate 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. Our NoHype system embodies four key ideas: (i) pre-allocation of processor cores and memory resources, (ii) use of virtualized I/O devices, (iii) minor modifications to the guest OS to perform all system discovery during bootup, and (iv) avoiding indirection by bringing the guest virtual machine in more direct contact with the underlying hardware. Hence, no hypervisor is needed to allocate resources dynamically, emulate I/O devices, support system discovery after bootup, or map interrupts and other identifiers. NoHype capitalizes on the unique use model in cloud computing, where customers specify resource requirements ahead of time and providers offer a suite of guest OS kernels. Our system supports multiple tenants and capabilities commonly found in hosted cloud infrastructures. Our prototype utilizes Xen 4.0 to prepare the environment for guest VMs, and a slightly modified version of Linux 2.6 for the guest OS. Our evaluation with both SPEC and Apache benchmarks shows a roughly 1% performance gain when running applications on NoHype compared to running them on top of Xen 4.0. Our security analysis shows that, while there are some minor limitations with current commodity hardware, NoHype is a significant advance in the security of cloud computing.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

Categories and Subject Descriptors

C.0 [**General**]: System Architectures; D.4.6 [**Software Engineering**]: Operating Systems—*Security and Protection*

General Terms

Security

Keywords

Secure Cloud Computing, Hypervisor Security, Attack Vectors, Virtualization, Multicore, Hardware Security

1. INTRODUCTION

Cloud computing is transforming the way people use computers. Cloud computing is also transforming how networked services are run. The provider of a service (the cloud customer) is able to dynamically provision infrastructure to meet the current demand by leasing resources from a hosting company (the cloud infrastructure provider). The cloud infrastructure provider can leverage economies of scale to provide dynamic, on-demand infrastructure at a favorable cost. The provider does this by utilizing virtualization where virtual machines from multiple customers share the same physical server. This multi-tenancy, where mutually distrusting customers lease resources from the same provider, underscores the need for a secure virtualization solution.

Because of its central role, the virtualization software is a prime target for attacks. Unfortunately, the virtualization layer is quite complex and forms a very large trusted computing base (e.g. Xen has ∼200K lines of code in the hypervisor itself ∼600K in the emulator, and over 1M in the host OS). The large code base is not suitable for formal verification – the best formal verification techniques available today are only able to handle around 10K lines of code [18]. Further, bug reports such as those listed in NIST's National Vulnerability Database [6] show the difficulty of shipping bug-free hypervisor code. A malicious VM can exploit these bugs to attack the virtualization software. Exploiting such an attack vector would give the attacker the ability to obstruct or access other virtual machines and therefore breach confidentiality, integrity, and availability of the other virtual machines' code or data.

Such vulnerabilities make many companies hesitant about moving to the cloud [14]. If not for the security threat of attacks on a vulnerable virtualization layer, computing in the cloud has the potential to be *more* secure than in private facilities. This is due to the very nature of the large and critical data centers run by the cloud infrastructure

providers. The level of physical security (cameras, personnel, etc.) found in these data centers is often cost prohibitive for individual customers.

Previous approaches to securing the virtualization software have limitations when applied to cloud computing. Approaches that minimize the hypervisor, e.g., [32], are not suitable for cloud computing infrastructures because they greatly reduce functionality. Approaches that introduce a new processor architecture, e.g., [10], cannot be deployed today. Approaches that add extra software to verify the integrity of the hypervisor, e.g., [9], add overhead, leave a small window open between the attack and the reactive defense, and do not protect against attacks that exploit bugs in the hypervisor for purposes other than injecting code or modifying the control flow (e.g., bugs which cause the hypervisor to crash).

In this paper we present our NoHype system that takes the novel approach of *eliminating* the hypervisor attack surface altogether. We remove the need for virtual machines to constantly interact with the hypervisor during their lifetime (e.g., by short-circuiting the system discovery and avoiding indirection). With NoHype, we (i) retain the ability to run and manage virtual machines in the same manner as is done today in cloud computing infrastructures, (ii) achieve this with today's commodity hardware, and (iii) prevent attacks from happening in the first place.

We previously proposed the high-level idea of NoHype in a position paper [16]. The previous work discussed how preallocation and virtualized I/O devices could be used to remove the hypervisor and motivated why this idea works in the setting of hosted cloud infrastructures. In this paper, we present the complete design, implementation, and evaluation of a working NoHype system on today's commodity hardware. This new work adds a significant level of depth through a complete prototype design and implementation, as well as new performance and security analysis which identifies and evaluates potential side-channels. In particular, we make the following new contributions:

- **An architecture which eliminates the hypervisor attack surface.** As we previously proposed [16], the architecture pre-allocates memory and processor cores (so a hypervisor does not have to do this dynamically) and uses only virtualized I/O devices (so a hypervisor does not have to emulate devices). New to this paper, the architecture also avoids the indirection of mapping physical interrupts and identifiers to the virtual ones (so a hypervisor does not have to be involved in any communication channels). Each of these is enabled by the unique use case of the cloud computing model.
- **A design which can be realized on today's commodity hardware.** We take advantage of modern hardware features to enforce the resource allocations as well as the common practice for cloud infrastructure providers to provide a set of kernel images. With this, we allow the guest operating systems to boot up as normal. This process is supported by a temporary hypervisor, which enables a slightly modified guest operating system to discover the environment at boot up and save this information about the system for later use.
- **A prototype implementation and system evaluation.** We built a NoHype prototype using an Intel Nehalem quad-core processor. We utilized Xen 4.0 to prepare the environment for the guest virtual machines, and a slightly modified version of Linux 2.6 for the guest operating system. Our performance evaluation through the use of SPEC 2006 [15] and Apache [1] benchmarks show roughly 1% faster run times. We also present a security analysis which shows that NoHype makes an improvement in the security of virtualization technology for hosted and shared cloud infrastructures.

The remainder of the paper is organized as follows. In Section 2 we discuss background information about virtualization. In Section 3 we discuss our threat model. In Section 4 we discuss the NoHype system architecture, while in Section 5 we discuss details of our prototype implementation. The security analysis is presented in Section 6. Finally, related works are discussed in Section 7 and we conclude in Section 8 including a discussion about future work where we discuss live virtual machine migration.

2. VIRTUALIZATION VULNERABILITIES

The purpose of a hypervisor is to present to the guest VM a view that appears as though the operating system and applications inside the VM are running directly on some given hardware. The hypervisor achieves this by emulating the underlying hardware and arbitrating access to it. Realizing this functionality requires a large and complex body of software. It also requires a significant and frequent interaction between the guest VMs and the hypervisor. This interaction is the basis of the security threat which a malicious VM can utilize to attack the hypervisor and exploit bugs in the hypervisor or supporting virtualization software to attack another VM.

2.1 Roles of Virtualization Software

Figure 1 shows the hypervisor, host OS, and emulator (e.g., QEMU) which form the virtualization layer. The dotted line outlines the components which are the trusted computing base in today's virtualization solutions. Highlighted in the figure are also the points of interaction and components which are involved in these interactions.

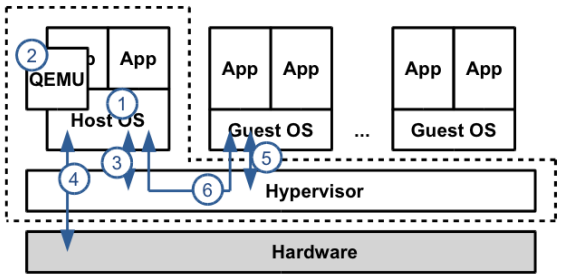


Figure 1: Roles of hypervisor.

An important and large part of the virtualization layer, in addition to the hypervisor, is the host OS (①) and the emulator (②). The host OS (Dom0 in Xen's terminology) has special management privileges and a system administrator uses the host OS to manage VMs on the system, e.g., launching and shutting down VMs. This causes the host OS to interact with the hypervisor via hypercalls (③). The

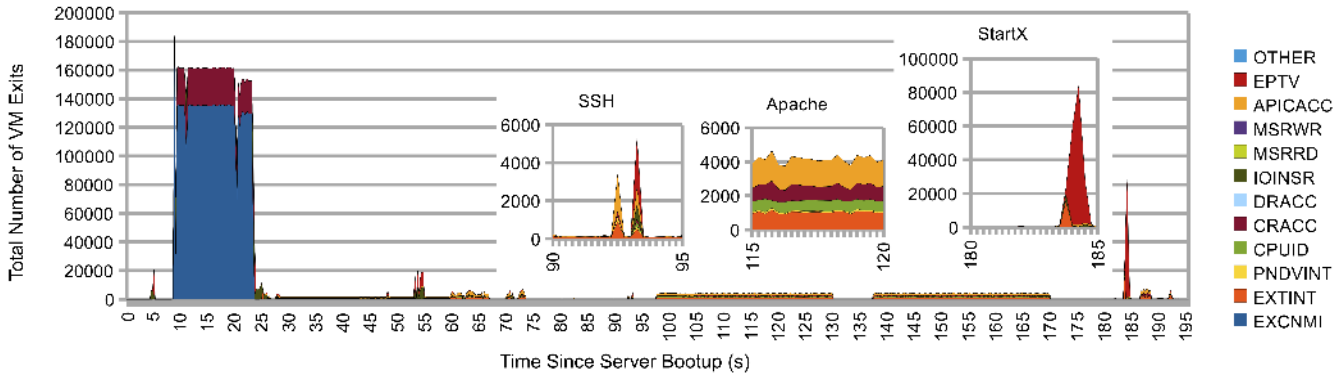


Figure 2: VM exits in stock Xen 4.0 during server bootup and runtime of a guest VM.

host OS may also include drivers for the physical devices which are used by the system (④). An emulator which emulates system devices that guest VMs interact with is also commonly run within the host OS. All together, this functionality requires over a million lines of code.

We now dicuss the roles of the hypervisor in emulating and managing the main resources in today's computer systems.

Processor cores: The first responsibility of any virtualization solution is the ability to arbitrate access to the processor cores as well as sometimes emulate their functionality. The hypervisor needs to cope with the dynamic addition and removal of VMs as they get started and stopped, as well as the ability to schedule multiple VMs on the same system. In addition to scheduling the VMs, the virtualization layer also uses the scheduler to periodically, every clock tick, run some of its own functionality – for example the hypervisor needs to check whether to deliver timer interrupts to the VM in this clock tick. The virtualization layer may also emulate the underlying processor as the presented processor cores may differ in capabilities from the actual cores.

Memory: The hypervisor is also in charge of managing the host physical memory of the system. Each VM is assigned some guest physical memory that the guest OS can manage and assign as virtual memory to the applications. Here, the virtualization layer can take a number of different approaches to managing the allocation of host physical memory among the various VMs. One popular use of virtualization is server consolidation which relies on the virtualization layer to dynamically manage the allocation of physical memory to the different VMs in the system.

I/O devices: Another important functionality of the virtualization layer is dealing with I/O devices such as a network interface card (NIC). Today's virtualization solutions commonly create virtual devices that they assign to the guest VMs. The virtualization layer needs to manage these virtual devices and emulate their behavior. There is also the option of dedicating physical devices to the VMs. In this situation, the virtualization layer only needs to assign and reclaim the devices. It does not need to emulate their behavior. In both cases, however, the emulator is needed to emulate the PCI (Peripheral Component Interconnect) bus. The guest OS interacts with the PCI bus during bootup to discover and configure the available devices.

Interrupts and Timers: The virtualization layer must also emulate the interrupt subsystem and timers for the guest VMs. For interrupts, the main parts of the underlying hardware interrupt subsystem emulated by the hypervisor include the I/O APIC[1] (which routes interrupts from devices to processor cores) and local APICs (which are attached to each core as an interface to the interrupt subsystem for sending and receiving interrupt messages to and form other local APICs or the I/O APIC). These APICs are emulated so the guest OS can use them to configure and manage interrupts inside the VM, while physical APICs are controlled by the hypervisor for the management of the physical platform. For timers, since the hypervisor utilizes the physical timer devices to get periodic clock ticks itself, it has to emulate the timer device for each guest VM. Today, VMs can be scheduled on different cores and as such, the interrupts and timers must first go through the hypervisor which will deliver the interrupts to the VM by utilizing its knowledge of the VM's current location.

2.2 Attack Surface

The virtualization layer is heavily involved throughout the lifetime of the guest VM. Each interaction between the VM and the hypervisor is then a potential attack vector that could be exploited by the guest. The guest OS interacts directly with the hypervisor (⑤) and indirectly with the host OS and the emulator (⑥) through VM exits (and hypercalls if paravirtualization is used). A VM exit is an event which occurs when the VM's code is interrupted and the hypervisor code begins to execute to handle some event (e.g., emulate memory access, deliver a virtual timer interrupt, etc.). A hypercall is similar to a system call and is used by guest VMs to request explicit service from the hypervisor. Hypercalls are not considered further as we are not using paravirtualization. VM exits are quite frequent even when the guest OS inside the VM is not doing any work; in an idle VM running on top of Xen 4.0, the VM exits occur ∼600 times per second.

On a 64-bit Intel x86 architecture with virtualization extensions, there are 56 reasons for VM exits, and this forms the large attack surface which is the basis of the security threat. The reasons VM for exits are described in Table 1. Each VM exit causes the hypervisor code to run so the hypervisor can intervene when the guest OS performs some operation that caused the associated exit. This allows the hypervisor to maintain the abstraction of the system which it presents to the guest OSes, for example it can return different CPUID values than the values actually reported by the hardware.

In order to emphasize how often the large and complex

[1]APIC is the advanced programmable interrupt controller.

virtualization layer is needed, we examine the execution of an actual (non-paravirtualized) VM with the Xen hypervisor. Figure 2 shows a timeline of a Linux based VM pinned to a core and with a directly assigned NIC, booting up and running some programs on top of the Xen 4.0 hypervisor with no other VMs present. The most frequent reasons for exits are highlighted in the figure while others are grouped in the "other" category. The stacked area graph shows the number of VM exits during each 250ms interval.

First, the guest OS boots up, which is the first 75 seconds in the graph. A staggering 9,730,000 exits are performed as the system is starting up. This is an average of about 130,000 exits per second. The majority of the exits are the EPTV and EXCNMI VM exits, but exits such as CPUID are also present. These exits are due to interaction with hardware that is being emulated (EPTV and EXCNMI) or are part of system discovery done by the guest OS (CPUID).

Next, at around 90 seconds we show an SSH login event which causes a spike in VM exits (EXTINT and APICACC). Even with the directly assigned NIC, the hypervisor is involved in the redirection of the interrupts to the correct core and the EXTINT exit signals when an interrupt has come. The APICACC exits are due to the guest interacting with the local APIC (Advanced Programmable Interrupt Controller) which is used to acknowledge receipt of an interrupt, for example.

Then, at around 115 seconds we show execution of the Apache web server with a modest request rate. This causes many exits: EXTINT, CPUID, CRACC, APICACC. We found that libc and other libraries make multiple uses of CPUID each time a process is started and in Apache there are new processes started to handle each connection.

Finally, at around 180 seconds we show starting VNC (a graphical desktop sharing software) and running `startx` to start the X Window System. We can see the EPTV exits which are due to emulation of VGA once the graphical window system is started.

To summarize, different situations lead to different uses of VM exits and invocation of underlying hypervisor support. Each VM exit could be treated as a communication channel where a VM implicitly or explicitly sends information to the hypervisor so the hypervisor can handle the event. Each VM exit is then a potential attack vector as it is a window that the VM can use to attack the hypervisor (e.g., by exploiting a bug in how the hypervisor handles certain events). With NoHype, we eliminate these attack vectors.

3. THREAT MODEL

With NoHype, we aim to protect against attacks on the hypervisor by the guest VMs. A malicious VM could cause a VM exit to occur in such a manner as to inject malicious code or trigger a bug in the hypervisor. Injecting code or triggering a bug could potentially be used to violate confidentiality or integrity of other VMs or even crash or slow down the hypervisor, causing a denial-of-service attack violating availability. We eliminate the need for interaction between VMs and hypervisor, thus preventing such attacks.

To that end, we assume the cloud infrastructure provider is not malicious, and sufficient physical security controls are being employed to prevent hardware attacks (e.g., probing on the memory buses of physical servers) through surveillance cameras and restricted access to the physical facilities. We are not concerned with the security of the guest OSes, but do require that a cloud provider makes available a set of slightly modified guest OS kernels which are needed to boot a VM. The responsibility of protecting software which runs *inside* the VMs is placed upon the customer. Also, in this work the security and correctness of the cloud management software is not covered. The cloud management software presents the interface that cloud customers use to request, manage and terminate virtual machines. It runs on dedicated servers and interacts with the NoHype servers. For the purpose of this paper, we assume it is secure, but will revisit this as potential future work.

Table 1: Selected reasons for VM exits [3].

VM Exit	Reason
EPTV	An attempt to access memory with a guest-physical address was disallowed by the configuration of the EPT paging structures.
APICACC	Guest software attempted to access memory at a physical address on the APIC-access page.
MSRWR	Guest software attempted to write machine specific register, MSR.
MSRRD	Guest software attempted to read machine specific register, MSR.
IOINSR	Guest software attempted to execute an I/O instruction.
DRACC	Guest software attempted to move data to or from a debug register
CRACC	Guest software attempted to access CR0, CR3, CR4, or CR8 x86 control registers.
CPUID	Guest software attempted to execute CPUID instruction.
PNDVINT	Pending virtual interrupt.
EXTINT	An external interrupt arrived.
EXCNMI	Exception or non-maskable interrupt, NMI.

4. NOHYPE SYSTEM ARCHITECTURE

In this section we present the NoHype system architecture which capitalizes on the unique use model of hosted and shared cloud infrastructures in order to eliminate the hypervisor attack surface. Rather than defending once these attack vectors have been utilized to attack the hypervisor, we take the new approach of removing the attack surface. In doing so, we have a goal of preserving the semantics of today's virtualization technology – namely that we can start VMs with configurable parameters, stop VMs, and run multiple VMs on the same physical server. Additionally, the NoHype architecture is designed to be realizable on today's commodity hardware. In subsequent sections we discuss the key ideas of the NoHype architecture which are:

- pre-allocating memory and cores,
- using only virtualized I/O devices,
- short-circuiting the system discovery process, and
- avoiding indirection.

4.1 Pre-allocating Memory and Cores

One of the main functions of the hypervisor is dynamically managing the memory and processor cores' resources toward the goal of overall system optimization. By dynamically managing the resources, VMs can be promised more resources than are actually physically available. This oversubscription is heavily used in enterprises as a way to consol-

idate servers. In a hosted cloud computing model, however, oversubscription is at odds with the expectations of the customer. The customer requests, and pays for, a certain set of resources. That is what the customer expects to receive and not the unpredictable performance and extra side channels often associated with oversubscription. Rather than relying on customers underutilizing their VMs, the cloud infrastructure provider could instead simply capitalize on the rapidly increasing number of cores and memory in servers to host more VMs per physical server (even without oversubscription). With NoHype we pre-allocate the processor cores and memory so that a hypervisor is not needed to manage these resources dynamically – this is possible in hosted cloud computing since the customer specifies the exact resources needed before the VM is created.

Today, the hypervisor dynamically manages the processor cores through the scheduling functionality. Since the number of cores is specified by the customer before the VM is created, in NoHype we dedicate that number of cores to the specific VM. This is not to be confused with pinning a VM (also called setting the processor core affinity of a VM), which is a parameter to the hypervisor scheduling function to configure the restrictions of which cores a given VM is allowed to run on. Additionally, today when the cores are not dedicated to a single VM and when the core which the VM is scheduled on can change, the hypervisor must emulate the local APIC by providing a timer and handling IPIs, which are functionalities that the guest OS would expect from a local APIC. Since with NoHype, a core is dedicated to a given VM, we no longer need to emulate this functionality and can allow a guest to use the local APIC directly. While this gives the guest OS direct control over the hardware local APIC and opens a new possibility for a guest VM to send a malicious interprocessor interrupt (IPI) to other VMs, we show in our security analysis in Section 6.2 that this can be handled with a slight modification to the way the guest VM handles IPIs.

Similar to pre-allocating processor cores, in NoHype we also pre-allocate the memory. In order to remove the virtualization layer, we can again capitalize on the use model where customers of the cloud infrastructure provider specify the amount of memory for their VMs. This means we can pre-allocate resources rather than having a hypervisor dynamically manage them. A key to isolating each virtual machine is making sure that each VM can access its own guest physical memory and not be allowed to access the physical memory of other VMs. Without an active hypervisor we must utilize the hardware to enforce the memory isolation by capitalizing on the hardware paging mechanisms available in modern processors.

4.2 Using only Virtualized I/O Devices

I/O devices are another important component that is addressed in the NoHype architecture. With NoHype, we dedicate I/O devices to the guest VM so we do not need virtualization software to emulate these devices. Of course, dedicating a physical I/O device to each VM does not scale. With NoHype, the devices themselves are virtualized. However, a VM running 'in the cloud' has no need for peripheral devices such as a mouse, VGA, or printer. It only requires network connection (NIC), storage, and potentially a graphics card (which is increasingly used for high-performance general-purpose calculations). So, only a limited number of devices with virtualization support is needed. Today, NICs with virtualization support are already popular and storage and graphics devices will be soon. Moreover, networked storage can be utilized in lieu of a virtualized (local) storage device – making the unavailability of virtualized storage devices only a minor limitation.

NoHype capitalizes on modern processors for both direct assignment of devices as well as virtualization extensions in modern commodity devices. VMs control the devices through memory-mapped I/O. The memory-management hardware of modern commodity processors can be configured so that the VM can only access the memory of the device that is associated with it (and the device can also only access the memory of its associated VM). Further, virtualization extensions are available in modern commodity devices. For example, SR-IOV[2] [7] enabled device announces itself as multiple devices on the PCI bus. The functionality of these devices is separated into board-wide functions (known as physical functions) which are controllable only by the host OS , and functions which are specific to the individual virtual devices (known as virtual functions) that can be assigned to different guest VMs.

4.3 Short-Circuiting the System Discovery

To run on a variety of platforms, most operating systems automatically *discover* the configuration of the underlying system. This is done by the guest OS kernel during its initial loading. To minimize changes to the guest OS, the NoHype architecture allows the guest OS to perform its normal bootup procedure, but slightly modifies it to cache system configuration data for later use. This is supported by a temporary hypervisor to overcome current limitations of commodity hardware. For example, to determine which devices are available, the guest OS will perform a series of reads to the PCI configuration space in order to determine what devices are present. This PCI configuration space, along with "system discovering" instructions such as CPUID, are not virtualized by today's hardware.

This modified guest OS kernel is provided by the cloud infrastructure provider – a practice that is common today to make it easier for customers to start using the cloud provider's infrastructure. This becomes a requirement in the NoHype architecture to ensure that no customer code executes while any underlying virtualization software is present – since the customer code may attempt to attack this temporary hypervisor. Additionally, as minimal changes are required to ensure all system information is discovered only during guest OS initialization, the cloud provider can make these minimal changes for the benefit of all of its customers. Importantly, this does not restrict what applications and guest OS kernel modules the customer can run, so restricting the customer's choice to a fixed set of guest OS kernels is not a significant problem.

Once the initial guest OS kernel bootup sequence is complete and the underlying system configuration has been learned by the guest OS, the temporary hypervisor is disabled. At this point the guest VM execution switches from code under the control of the cloud infrastructure provider to the customer's code which can run any applications and load any guest OS kernel modules desired.

In addition to supporting the device discovery and system information instructions, the guest OS will utilize one

[2]SR-IOV is the Single-Root I/O Virtualization specification.

additional device during its initialization. In particular, a high precision timer, such as the high-precision event timer (HPET), is needed temporarily during the boot process of the Linux kernel. First, it is used as an external reference to determine the clock frequency of the processor. The local APIC timer is configured to generate an interrupt in a certain number of clock cycles (rather than certain time). Therefore, it is important to know the exact frequency of the processor clock so the local APIC can be used as an accurate timer[3]. Second, this high precision timer is used to emulate the CMOS-based real-time clock – that is, the battery backed clock which tells the date and time. Again, the temporary hypervisor can emulate this functionality to let the guest discover the processor core's frequency and get the time of day. After this, the clock is not needed anymore[4].

Finally, since we allow this system discovery only during bootup, the OS must be sure to gather all of the information that may be needed over the lifetime of the VM. We capitalize on the fact that we are providing the guest OS kernel by making minor modifications so that the information is gathered during guest OS bootup and cached by the OS. This removes the need for instructions like the CPUID to run during the lifetime of the OS to determine hardware configuration, and therefore the need for a hypervisor response.

4.4 Avoiding Indirection

Today, because hypervisors present an abstracted view of a machine that is not a one-to-one mapping of virtual to real hardware, they must perform indirections that map the virtual view to real hardware. Since we are bringing the guest virtual machine in more direct contact with the underlying hardware, we avoid these indirections, and therefore, remove the need for a hypervisor to perform them.

One such indirection is involved in the communication between cores. Hypervisors present each VM with the illusion of running on a dedicated system. As such, the hypervisor presents each VM with processor IDs that start at 0. Today, the physical cores in the processor can be shared by more than one VM and the core that a VM is running on can be changed by the hypervisor's scheduler. Because of this, the hypervisor needs a map between the view presented to the VM and the current configuration in order to support communication between the VM's virtual cores. When dedicating cores to VMs, as is the case with NoHype, the guest VM can access the real processor ID and avoid the need for indirection.

Indirection is also used in delivering interrupts to the correct VM. For the same reason that the processor core ID requires indirection (VMs can share cores and can move between cores), the hypervisor has to handle the interrupts and route them to the correct VM. When dedicating cores to VMs, we remove the need for the re-routing as interrupts go directly to the target VM.

5. PROTOTYPE DESIGN

In this section we present the prototype of our NoHype system. Rather than write from scratch all of the necessary software to setup and boot a guest virtual machine, we instead utilize existing virtualization technology which must also provide this functionality. We base our prototype off of Xen 4.0, Linux 2.6.35.4 as the guest OS, and an Intel XEON W5580 processor. For the virtualized network card we used one with the Intel 82576 Ethernet controller. We utilized networked storage instead of a virtualized disk, since there are no commercially available drives which support SR-IOV at this time. In particular we used iPXE[5] for a network boot to fetch the guest OS kernel along with iSCSI [27] for the guest VM's storage.

In order to understand the details of what changes were required to the various software components, it is useful to understand what is happening during the various phases of execution. Shown in Figure 3 is a time line of a guest VM's life time – from creation to shutdown. The following subsections will detail each phase and discuss the changes we made to Xen or Linux to support that phase. We will wrap up the section by presenting an evaluation of the raw performance improvements seen with our prototype.

5.1 VM Creation

Independent of the interface that is presented to the customer for managing virtual machines, eventually, a request from the customer will result in a request sent by the cloud management software to the system software running on a specific server to create a virtual machine. This request will specify all of the configuration details, such as the amount of memory, the number of cores, and what (virtual) devices to assign to the VM.

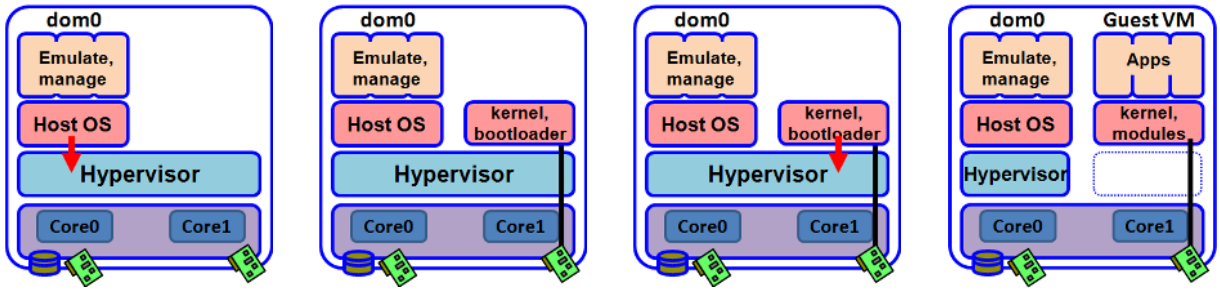
During this phase all of the resources are pre-allocated and the virtualized I/O devices are assigned. Here, Xen already provides all of the required functionality. The management software runs in Dom0, Xen's privileged VM, and we restricted it to execute only on core 0, as shown in Figure 3(a). The VM initialization code then configures the hardware mechanisms which will enforce memory allocation – in particular, the extended page tables (EPT) in the Intel processors. With NoHype, we require that these tables be preset so that there is no need for a hypervisor which manages memory translation dynamically. Xen's VM initialization code already has such pre-setting of EPT entries for this purpose. The VM initialization code also includes the physical function driver for the NIC which sets up the registers in the device not accessible to guest VMs – e.g., the MAC address, multicast addresses, and VLAN IDs.

For pre-allocating processor cores, Xen's VM initialization code has the ability to pin a VM to a set of cores. It does this by setting the processor core affinity of the VM which causes the scheduler function to re-schedule the VM on the selected core and add it to the list of VMs for which it can choose between for that core. Note that while pinning sets which cores a given VM can run on, it does not restrict the pinning of multiple VMs to the same core. For NoHype, the management software needs to keep track of which cores are already assigned and only pin VMs to unused cores.

Finally, Xen's VM initialization code allocates the virtualized NIC(s) via the PCI pass through mechanism supported in Xen. That is, it sets EPT entries to enable the device's memory range to be mapped to the VMs memory space. It also utilizes the VT-d [4] extensions in the Intel architecture to allow the device to DMA directly into the VM's memory.

[3]In newer processors the local APIC runs at a fixed frequency regardless of the processor core's idle states [2] so it can be used to keep track of processor ticks.

[4]The network time protocol can be used to ensure the clock stays accurate.



(a) Creation. (b)Bootup. (c) Disengagement (d) Execution/Shutdown.

Figure 3: The four stages of a VM's lifetime in a NoHype system.

5.2 Guest VM bootup

Once the VM is created, its boot process is kicked off, as seen in Figure 3(b). We piggyback on Xen's inclusion of a bootloader called "hvmloader" (hvm stands for hardware virtual machine, which indicates the processor has certain virtualization capabilities). The hvmloader is the first software that executes inside the guest VM. It has network boot capabilities through the inclusion of iPXE[5] which enables it to fetch, in our case, the guest OS kernel and initial RAM disk[5]. Once the guest OS kernel is loaded and ready to boot, the hvmloader sets the guest OS kernel parameters to pass information to the guest OS and jumps to the kernel.

During the operating system bootup, the guest OS kernel will perform a great deal of system discovery – in particular, device discovery as well as discovering the processor capabilities. We discuss each of these in further detail below. Recall that during this phase, a temporary hypervisor is present to support the system discovery.

5.2.1 *Discovering Devices*

In order to determine what devices are present, in particular PCI devices, the guest operating system queries a known range of memory addresses. If the response to the read of a particular memory address returns a well-known constant value, that means there is no device present at that address. Otherwise the device would return a device identifier. In Xen based VMs, reads to these addresses trap to the hypervisor. Xen then passes the request to QEMU running in Dom0 which handles it. In QEMU today, there is an assumption of a minimal set of devices being present (such as VGA). We modified QEMU to return "no device" for all but a network card.

Upon discovering the device, the guest OS then sets up the interrupts for that device by choosing vectors and setting up internal tables that associate the vectors with interrupt handler functions. When the guest OS configures the I/O APIC with this information, the request traps to the hypervisor which virtualizes the I/O APIC in software. Since the guest's view of available vectors does not match what is actually available, Xen chooses a vector which is free, and stores a mapping between the actual vector and what the guest expects. This means that Xen would typically handle the interrupt, perform a mapping, and then inject an interrupt with the mapped vector. However, since we will eventually be disengaging the hypervisor, we modified both Xen and the guest Linux kernel to make the vector chosen by each to be configurable. Linux is made configurable so that it chooses a vector which is actually available and so that it does not choose the same vector as another VM. Xen is made configurable so that the management software can ensure that the Xen vector assignment function will also choose this vector. Once the interrupts are set up, the guest OS sets up the device itself through the virtual function driver's initialization routine. In the particular NIC we used for our prototype, part of this initialization utilizes a mailbox mechanism on the NIC for interacting with the physical function driver in the host OS to perform a reset and retrieve the MAC address. After the virtual function driver is initialized, interaction with the physical function driver is not needed.

[5]The release version of Xen includes gPXE. iPXE is a more actively developed branch and one for which we added a driver for the Intel 82576 Ethernet controller.

5.2.2 *Discovering Processor Capabilities*

In addition to needing to know which devices are available, the guest OS needs to know details about the processor itself – in particular, (i) the clock frequency, (ii) the core identifier, and (iii) information about the processor's features.

The frequency that the processor runs at must be calculated from a reference clock. For this, we provide a high precision event timer (HPET) device to the guest VM. Since this device is not virtualized in hardware, we only have a software virtualized HPET providing the guest VM with periodic interrupts during bootup when the operating system will need it. Once the operating system knows the clock frequency of the core, it can use the per-core local timer as its timer event source rather than the HPET.

The core identifier is used so that when the software running on one core wants to send an interprocessor interrupt (IPI) to another core, it knows what to set as the destination. In Xen, this identifier is assigned by Xen and any IPIs involve a mapping within the hypervisor to the actual identifier. In order to unwind this indirection, we modified Xen to pass the actual identifier of the core, which in Intel processors is the local advanced programmable interrupt controller (APIC) ID. This identifier can be obtained by the guest operating system in three ways, each of which we modified. First, it can be obtained in the *APIC ID* register within the local APIC itself. Second, it can be obtained through the CPUID instruction by setting the EAX register to '1'. Finally, it can be obtained with the Advanced Configuration and Power Interface (ACPI) table, which is written by the bootloader (hvmloader for Xen) as a way to pass information to the operating system.

Finally, information about the processor's features such

as cache size and model number, are obtained through the CPUID instruction. This is an instruction that applications can use in order to do some processor-specific optimizations. However, in a virtual environment the capabilities of the processor are different than the actual capabilities, and therefore when the instruction is encountered, the processor causes an exit to the hypervisor which emulates the instruction. We modified the Linux kernel to perform this instruction during boot-up with each of the small number of possible input values, storing the result for each. We then make this information available as a system call. Any application that calls the CPUID instruction directly will have to be modified so they do not cause a VM exit. While this may sound like a major hurdle, in reality, it is not. We did not encounter any such applications, but instead encountered the use of the CPUID instruction in a small number of standard libraries such as libc which calls CPUID whenever a process is created. We modified libc to use the system call instead of the instruction. While these are not part of the guest OS kernel (and therefore not provided by the cloud provider), they can be made available for customers to easily patch their libraries and do not require a recompilation of the application. Further, for any application which does make use of CPUID and cannot be recompiled or modified, simple static binary translation can be used to translate the CPUID instruction into code which performs the system call and puts the results in the expected registers.

5.3 Hypervisor Disengagement

At the end of the boot process, we must disengage the hypervisor from any involvement in the execution of the guest VM. We achieve this through a guest OS kernel module which is loaded and unloaded within the init script of the initial RAM disk (initrd). As shown in Figure 3(c), this module simply makes a hypercall with an unused hypercall number (Dom0 communicates with the hypervisor through hypercalls). The handler for this particular hypercall will perform the hypervisor disengagement for that core and send an IPI to the other cores allocated to the VM to signal to them that they need to perform the core-specific disengagement functionality.

There are three main functions of this process. First, it must take steps to remove the VM from several lists (such as timers which are providing timer interrupts to the guest) as well as remove the set of cores from the online processor cores mask (so Xen does not attempt to send any IPIs to it). Second, the disengagement function must configure the hardware such that the guest has full control over the individual core. This includes settings within the virtual machine control structure (VMCS)[6] (e.g., setting the virtualize APIC access bit to 0) as well as mappings in the extended page tables (e.g., adding the local APIC's memory range so that it does not cause an EPT violation). Finally, we must initialize the local APIC registers with values that match what the guest operating system wrote. Before this disengagement, Xen makes use of the local APIC itself and presents a virtualized view to the guest (e.g., Xen uses a one-shot timer as its own timer source but presents a periodic timer to our Linux kernel). The disengagement function sets the registers in the actual local APIC with the values that are stored in the virtualized local APIC.

[6]The VMCS is used to manage transitions between the guest virtual machine and the hypervisor.

Once the hypervisor disengagement process completes, execution returns to the guest VM where the disengagement-initiator module is unloaded and the iSCSI drive(s) for the customer is mounted. Execution control is then transferred (for the first time) to the user's code.

5.4 Guest Execution and Shutdown

At this point, execution is under the complete control of the guest, as shown in Figure 3(d). It can run applications and load OS kernel modules. We have configured the system such that anything the virtual machine may need to do, it will be able to do. We consider any other actions to be illegal (and potentially malicious). Many actions, such as accessing memory outside of the allocated memory range, will cause an exit from the VM. Since we consider them illegal, they will result in the termination of this VM. Other actions, such as sending messages to the physical function driver via the mailbox functionality on a device, can be ignored.

Because of this restriction, we needed to modify the guest Linux kernel slightly – these modifications do not affect an application or kernel module's interaction with Linux. That is not to say we must trust the guest OS for the security of the entire system, simply that in order for the guest VM to run without causing a VM exit, Linux is configured to not access devices that it is not using. In particular, Linux assumes the presence of a VGA console and writes to the fixed I/O port whether there is a VGA console or not. We modified this assumption and instead made the use of a VGA console configurable. Additionally, Linux makes the assumption that if an HPET device is available for determining the clock frequency, it should be added to the list of clock sources. Each of the clocks in this list are periodically queried for its cycle count. As we have a time stamp counter (TSC) also available, the HPET is not needed. We added a configuration in Linux to specify whether the HPET device is to be added to the list of clock sources or not.

However, one limitation of completely removing the availability of an HPET device is that we are now relying on the local APIC timer. In processors before the current generation, this local APIC timer would stop when the core goes into a deep idle state. Recent processors with the Always Running APIC Timer (ARAT) feature are not subject to this limitation. Unfortunately, we built our prototype on a processor without this feature. To overcome this, rather than buying a new processor, we simply faked that we have it by (i) specifying that the processor has the ARAT capability in the response to the CPUID instruction, and (ii) using the Linux parameter, *max_cstate*, to tell Linux to not enter a deep idle state (clearly not ideal, but acceptable for a prototype).

Our timeline ends when the guest VM shuts down. A guest VM can initiate the shutdown sequence itself from within the VM. This will eventually result in an exit from the VM, at which point the VM is terminated. However, we cannot rely on customers to cleanly shutdown when their time is up. Instead, we need to be able to force the shutdown of a VM. We realize this by configuring the VMCS to specify that the VM should exit when the core receives a non-maskable interrupt (NMI). In this way, the hypervisor, restricted to running core 0 at this point, can send an NMI, forcing a VM exit, giving the VM exit handler the ability to shutdown the VM.

5.5 Raw Performance Evaluation

Our prototype was built as a demonstration that we can actually realize NoHype on today's commodity hardware. In addition to the security benefits, which we analyze in Section 6, removing a layer of software leads to performance improvements, since with NoHype, there will no longer be the number of VM exits as seen in Figure 2.

We experimented with both the SPEC benchmarks which analyze the performance of the system under different workloads as well as a VM running Apache to capture a common workload seen in cloud infrastructures today. In each case we ran the experiment with both NoHype and stock Xen with a hardware virtual machine guest. With the NoHype system, we utilized our modified Linux kernel, whereas in the Xen system we utilized the unmodified 2.6.35.4 Linux kernel. Each VM was configured with two cores, 4GB of memory, and two network cards that were passed through (one an Internet facing NIC, and one for communicating with the iSCSI server). There was no device emulation and no other VMs were running on the system which might interfere with performance.

Shown in Figure 4 are the results of our experiments. We saw an approximately 1% performance improvement across the board. The lone exception to this was the gcc benchmark, which saw better performance with Xen than with NoHype. We need to further investigate the cause of this, but believe it to be related to our modifications to the guest kernel. Also note that much of the major performance bottlenecks associated with virtualization are alleviated with the VT-d (to directly assign devices) and EPT (to allow VMs to manage page tables) technologies and therefore already used in Xen. Our performance improvement comes from removal of the VM exits and is on top of performance gained from using VT-d and EPT hardware.

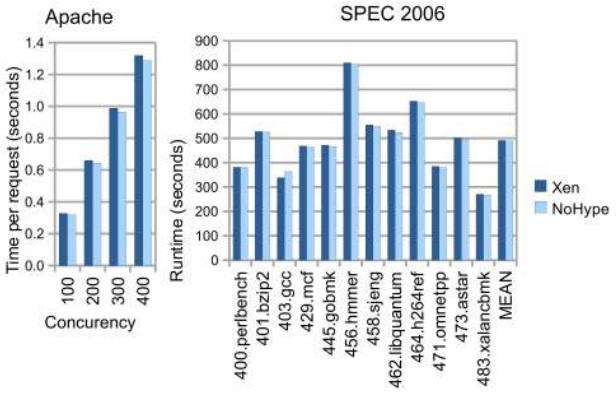


Figure 4: Raw performance of NoHype vs. Xen.

6. SECURITY ANALYSIS

In this section we present a security analysis of our NoHype architecture and its realization on today's commodity hardware. Our conclusion is that NoHype makes an important improvement in the security of virtualization technology for hosted and shared cloud infrastructures, even with the limitations due to today's commodity hardware.

6.1 Remaining Hypervisor Attack Surface

A NoHype system still requires system management software (performing some of today's hypervisor's duties) to be running on each server. While defending the interaction between the cloud manager and the system manager is our future work, here we have concentrated on the surface between the guest VM and the hypervisor which is much larger and more frequently used.

To initialize the guest VM, we use a temporary hypervisor and a slightly modified guest OS for performing system discovery tasks. The initial guest OS kernel is supplied and loaded by the cloud infrastructure provider, thus the customer has no control over the OS kernel which interacts with the temporary hypervisor. The temporary hypervisor is disabled (i.e., the VM is disengaged) before switching to the customer's code. By the time the customer's code runs, it does not require any services of the hypervisor (the system discovery data is cached and the devices, memory and cores are assigned). Any VM exit will trigger our kill VM routine as previously described, thus denying a malicious customer the opportunity to use a VM exit as an attack vector.

The ability of the guest VM to do something illegal, cause a VM exit, and trigger system management software to take action is itself a possible attack vector. The code handling this condition is in the trusted computing base (TCB) of a NoHype system, however, it is quite simple. After the VM is disengaged we set a flag in memory indicating that any VM exit should cause the VM to be terminated. The code base found in the temporary hypervisor and privileged system manger is never triggered by the guest VM after disengagement. Similarly, it cannot be triggered by any other running VM, as an exit from that VM will only trigger the kill VM routine.

6.2 VM to VM Attack Surface

After disengaging a VM, we give it full access to interprocessor interrupts (IPIs). One limitation of today's hardware is that there is no hardware mask which can be set to prevent a core from sending an IPI to another core. This introduces a new but limited ability for VM to VM communication. Now the VM has the ability to send an IPI to any other core without that core being able mask it or even know who sent it. The system management software which is pinned to one of the cores may also be a target of such an attack.

A preliminary pragmatic solution is presented here. Since the IPI does not contain any data and is only used as a trigger, the guest OS can defend against this attack by slightly modifying the way it handles IPIs. For each type of IPI, a shared region in memory can hold a flag that can be set by the sending core and then checked and cleared by the receiving core. Given that no VM can access memory of another VM, an attacker will not have the ability to set any of these flags. Therefore, the receiver of the IPI can simply ignore the IPI if the flag is not set.

While this ability for guest VMs to send IPIs poses very little security risk, it has the potential to enable an attacker to launch a denial of service attack by constantly sending IPIs to a given core. Fortunately, the extent to which they can possibly degrade performance is extremely limited. We set up an experiment by configuring an attacker VM with up to 8 cores, each sending IPIs at their maximum rate, and a victim with 2 cores. The results, shown in Figure 5, show that the performance degradation is limited to about 1%. Note that while we experimented with 8 cores, using 4 attacker cores was sufficient to saturate the rate at which IPIs can be sent in our test system. We used the Apache bench-

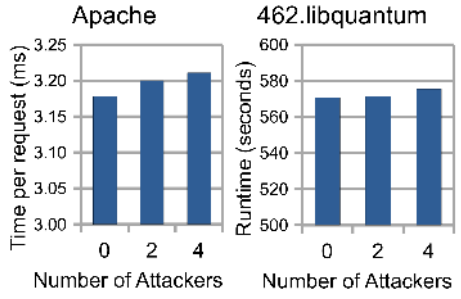


Figure 5: Effect of IPI attack on benchmarks.

mark and a compute intensive benchmark (462.libquantum) from the SPEC 2006 suite. Because the bus used to deliver IPIs is the same bus used for interrupts (the APIC bus), the performance of Apache was affected slightly more because it is a more interrupt driven application. The 462.libquantum benchmark is compute intensive and captures the overhead of simply having to process each interrupt.

6.3 Isolation between VMs

With NoHype, we rely on hardware mechanisms to provide isolation of access to shared resources. Most important are the confidentiality and integrity of each VM's memory. We utilize Xen's code which pre-sets the entries in the extended page tables (EPT) to assign physical pages to a particular VM. Any access to memory by a VM will cause the processor hardware to perform a translation using the EPT. When accessing memory outside of the given VM's allowed pages, a violation will occur and the VM will exit to the hypervisor context (which after disengagement is our kill VM routine). Because of this, the confidentiality and integrity are tied to the correctness of the hardware EPT implementation, which we believe will have undergone significant testing and verification. While all modern hypervisors also use the EPT mechanism, they may update the mapping when they provide additional capabilities such as transparent page sharing among VMs [8]. Consequently, isolation today depends not only on the hardware's correctness but also on the complex hypervisor software.

Also of importance is the performance isolation between VMs in the face of resource sharing. Without a hypervisor, as in NoHype, hardware is relied on to provide the isolation. The main shared resources of concern are the network card (and associated bandwidth) as well as the memory bus. The network card has queues with flow control mechanisms that provide a fair allocation of resources. The memory controller and bus, on the other hand, do not in today's processors [28].

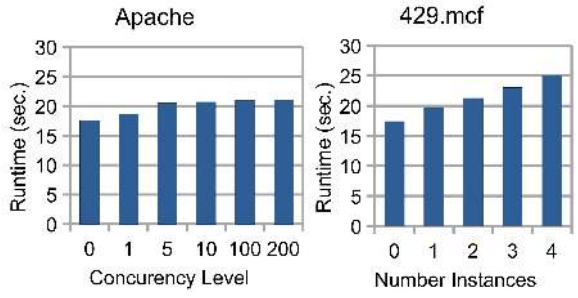


Figure 6: Quantification of the memory side-channel.

To quantify the quality of memory isolation, we ran one VM with a varying workload and examined the performance of a second VM with a fixed workload. In particular, the fixed workload VM had the memory intensive 429.mcf benchmark from SPEC 2006 suite running. In the varying workload VM we experimented with both Apache, under varying number of concurrent requests, as well as varying the number of instances of the 429.mcf benchmark. The results are shown in Figure 6. This experiment can be viewed either in terms of (i) an attacker attempting to affect the performance of a victim, or (ii) an attacker attempting to learn information about the victim (i.e., utilize it as a side channel to answer questions such as "how loaded is my competitor"). In either case, there is some interference between each workload, but we believe the interference is not significant enough to completely deny service or to learn sensitive information such as cryptographic keys [17].

6.4 VMs Mapping Physical Infrastructures

Work by Ristenpart, et al., [30] has raised the concern of infrastructure mapping attacks. With NoHype, the guest VMs have a more direct interaction with the hardware and could abuse that to try to map an infrastructure. One example may be a malicious VM reading the APIC ID numbers to identify the underlying physical cores and use that information to help narrow down where in the provider's infrastructure the VM is located. This may be mitigated by randomizing APIC IDs of the cores (which can be done at system boot time). Even if a malicious VM is able to determine that it is co-located with a victim VM, our approach of eliminating the attack surface denies it the opportunity to attack the hypervisor and by extension the victim VM.

7. RELATED WORK

The related work can be categorized in four main areas: minimizing the hypervisor, proposing a new processor architecture, hardening the hypervisor, or giving VMs more direct access to hardware.

Minimizing the hypervisor: Work on minimizing hypervisors aims to reduce the amount of code within the hypervisor, which should translate to fewer bugs and vulnerabilities. One example is SecVisor [32], a hypervisor which supports a single guest VM and protects that VM from rootkits. Another example is TrustVisor [26] which is a special-purpose hypervisor for protecting code and data integrity of selected portions of the application. Previous minimal hypervisors are not practical for deployment in the hosted cloud computing model where multiple VMs from multiple customers run on the same server. With NoHype, we show how to remove attack vectors (in effect also reducing the hypervisor) while still being able to support the hosted cloud computing model.

New processor architectures: In another approach, researchers propose building new processor architectures which explicitly offer new hardware functionality for improving security. Much work has been done on new hardware mechanisms for protecting applications and trusted software modules [24, 34, 21, 12], including Bastion [10] which uses a full hypervisor. Unfortunately, such approaches do require new microprocessors and cannot be deployed today, unlike our solution. Additionally, the use model for cloud computing has some similarities with that of mainframes. The architectures targeting these systems, such as the IBM Sys-

tem z [29], have support for creating logical domains which enforce partitioning of resources such as CPU and memory in hardware. In contrast, NoHype focuses on commodity x86 servers used by cloud infrastructure providers.

Hardening the hypervisor: Much of hypervisor-related work has centered around hardening of the hypervisor, such as [23, 31, 33]. Especially interesting is HyperSafe [35] which aims to protect a hypervisor against control-flow hijacking attacks. They use a non-bypassable memory lockdown technique (only a special routine in the hypervisor can write to memory) coupled with a restricted pointer indexing technique (all function calls in the hypervisor are transformed to jumps from a special table). While making it more difficult to subvert the hypervisor, these additions add about a 5% performance overhead and any bugs in the hypervisor could still be exploited through one of the attack vectors. Recently, HyperSentry [9] used the SMM (system management mode) to bypass the hypervisor for integrity measurement purposes. Unfortunately, the integrity measurements only reveal traces of an attack after it has already happened and are limited to protecting against attacks which persistently modify the hypervisor executable. While the authors report being able to invoke the measurement every 8 seconds in HyperSentry, this still leaves a window for attackers. Furthermore, their approach results in a 2.4% overhead if HyperSentry protections are invoked every 8 seconds. In contrast, NoHype prevents the attacks from happening in the first place, and does this with about a 1% performance improvement.

Direct access to hardware: NoHype shares much with exokernels such as ExOS [13] and Nemesis [22] which essentially reduce an operating system to providing only arbitration to shared resources and give applications more direct access to hardware. We capitalize on modern hardware advances to push it even further where the thin software layer from the exokernels is realized in hardware and full commodity operating systems can be run, rather than requiring applications to be redesigned for the exokernel environment. There have also been proposals that enable a single operating system to run without a virtualization layer but can insert a virtualization layer when needed – e.g., to run a second VM during planned maintenance [25] or to utilize migration for consolidation [19]. NoHype, on the other hand, can run multiple VMs simultaneously, each with direct access to its allocated hardware.

8. CONCLUSIONS AND FUTURE WORK

Today, the hypervisor is the all-powerful system software layer which controls the resources of a physical system and manages how they interact with the guest VMs. Because of its central role, the hypervisor, and other parts of the virtualization software, is a potent target for attacks, especially in shared infrastructures which allow multiple parties to run virtual machines. In this paper, we presented the complete design, implementation and evaluation of a working NoHype system on today's commodity hardware which removes the attack surface of the hypervisor and thus eliminates the vector by which VMs can exploit vulnerabilities. We do this by eliminating the VM's need for a hypervisor through (i) pre-allocation of processor cores and memory resources, (ii) using only virtualized I/O devices, (iii) supporting the system discovery process with a temporary hypervisor and a slightly modified guest OS, and (iv) avoiding any indirection that would necessitate having a hypervisor. This allows us to remove the interaction between the guest VMs and hypervisor and eliminate the attack surface which a malicious VM could use to compromise the virtualization layer, and then in turn attack or obstruct other VMs. In addition, our evaluation with benchmarks showed about 1% faster run times.

While NoHype significantly advances the security of shared cloud infrastructures, today's commodity hardware imposes some limitations; as future work, we will examine minimal hardware changes to further tighten the security of a NoHype system. Also, we will add support for live VM migration, particularly for the scenario where the initiator of the migration (the cloud provider) differs from the owner of the VM (the cloud customer). The disruption this process causes to the customer's VM depends on the workload of the VM [11], yet the provider does not know the workload or whether it is a particularly bad time to disrupt the guest VM. We believe the correct model for migration is for the provider to notify the customer ahead of time, allowing the customer to prepare for the transient disruption (e.g., by shedding load or redirecting new requests) and participate in the migration itself (e.g., through support for OS migration [20] in the guest). Finally, we plan to explore ways for the customer to run virtualization software of its own to enable nested virtualization, which may also aid in supporting live migration.

9. ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation grants: EEC-0540832 and CCF-0917134. Eric Keller was supported through an Intel Ph.D. Fellowship. We also benefited from equipment donation from Intel.

We would like to thank Tim Deegan from Citrix, Andrew Warfield from University of British Columbia, and Don Banks from Cisco, for discussions and feedback on our NoHype design. Additionally, we would like to thank our shepherd, Herbert Bos, and the anonymous CCS reviewers for their comments and suggestions.

10. REFERENCES

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M, page 274. `http://www.intel.com/products/processor/manuals/`.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `http://www.intel.com/products/processor/manuals/`.

[4] Intel Corporation: Intel Virtualization Technology for Directed I/O. `http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf`.

[5] iPXE: Open Source Boot Firmware. `http://ipxe.org/`.

[6] National Vulnerability Database, CVE and CCE Statistics Query Page. `http://web.nvd.nist.gov/view/vuln/statistics`.

[7] PCI SIG: PCI-SIG Single Root I/O Virtualization. `http://www.pcisig.com/specifications/iov/single_root/`.

[8] Understanding Memory Resource Management in VMware ESX Server. VMWare White Paper. 2009. www.vmware.com/files/pdf/perf-vsphere-memory_management.pdf.

[9] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pages 38–49, October 2010.

[10] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Jan. 2010.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[12] J. Dwoskin and R. B. Lee. Hardware-rooted trust for secure key management and transient trust. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles (SOSP)*, December 1995.

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

[15] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[16] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *International Symposium on Computer Architecture (ISCA)*, June 2010.

[17] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Computer Security: ESORICS 98*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. 1998.

[18] 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 *Symposium on Operating Systems Principles (SOSP)*, pages 207–220, October 2009.

[19] T. Kooburat and M. Swift. The best of both worlds with on-demand virtualization. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[20] M. A. Kozuch, M. Kaminsky, and M. P. Ryan. Migration without virtualization. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2009.

[21] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *International Symposium on Computer Architecture (ISCA)*, June 2005.

[22] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7), Sept. 1996.

[23] C. Li, A. Raghunathan, and N. K. Jha. Secure virtual machine execution under an untrusted management OS. In *Proceedings of the Conference on Cloud Computing (CLOUD)*, July 2010.

[24] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[25] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[26] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, May 2010.

[27] K. Z. Meth and J. Satran. Design of the iSCSI protocol. In *IEEE Symposium on Mass Storage Systems*, April 2003.

[28] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of USENIX Security Symposium*, August 2007.

[29] L. Parziale, E. L. Alves, E. M. Dow, K. Egeler, J. J. Herne, C. Jordan, E. P. Naveen, M. S. Pattabhiraman, and K. Smith. Introduction to the new mainframe: z/VM basics, Nov. 2007. http://www.redbooks.ibm.com/redbooks/pdfs/sg247316.pdf.

[30] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[31] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. V. Doorn, J. L. Griffin, S. Berger, R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. Doorn, J. Linwood, and G. S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, 2005.

[32] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41(6):335–350, December 2007.

[33] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *European Conference on Computer Systems*, April 2010.

[34] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *International Symposium on Computer Architecture (ISCA)*, June 2005.

[35] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.