Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering

Anh Nguyen[†], Himanshu Raj[*], Shravan Rayanchu[‡], Stefan Saroiu[*], and Alec Wolman[*]

[†]UIUC, [‡]University of Wisconsin, and [*]Microsoft Research

Abstract: *The set of virtual devices offered by a hypervisor to its guest VMs is a virtualization component ripe with security exploits – more than half of all vulnerabilities of today's hypervisors are found in this codebase. This paper presents Min-V, a hypervisor that disables all virtual devices not critical to running VMs in the cloud. Of the remaining devices, Min-V takes a step further and eliminates all remaining functionality not needed for the cloud.*

To implement Min-V, we had to overcome an obstacle: the boot process of many commodity OSes depends on legacy virtual devices absent from our hypervisor. Min-V introduces delusional boot*, a mechanism that allows guest VMs running commodity OSes to boot successfully without developers having to re-engineer the initialization code of these commodity OSes, as well as the BIOS and pre-OS (e.g., bootloader) code. We evaluate Min-V and demonstrate that our security improvements incur no performance overhead except for a small delay during reboot of a guest VM. Our reliability tests show that Min-V is able to run unmodified Linux and Windows OSes on top of this minimal virtualization interface.*

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Security kernels

1. Introduction

Cloud providers rely on commodity virtualization systems to enforce isolation among their customers' guest VMs. Any vulnerability that could be exploited by a guest VM has serious consequences for the cloud because it can lead to corrupting the entire cloud node or to launching DoS attacks on other guest VMs. We manually examined all such vulnerabilities found in a few online security databases [25, 27, 34, 40] and we discovered 74 such vulnerabilities in Xen, VMWare ESX, and VMWare ESXi, combined. Over 70% of them (52 security vulnerabilities) were present in these systems' virtualization stacks, that is in the code implementing the virtualized I/O offered to each guest VM. Table 1 presents a breakdown of these findings.

	Xen	VMWare ESX	VMWare ESXi	Total
# of security vulnerabilities in entire codebase	31	23	20	74
# of security vulnerabilities in virtual devices codebase	20	17	15	52
# of security vulnerabilities found in devices Min-V removes	**16**	**15**	**13**	**44**

Table 1. Security Vulnerabilities in Commodity Virtualization Systems. *These vulnerabilities were collected from four online databases [25, 27, 34, 40].*

These statistics suggest that minimizing the codebase implementing the virtualization stacks of these systems can go a long way towards preventing attacks from rogue VMs. We argue that the nature of cloud computing lends itself to making these virtualization stacks much smaller and thus more safe. First, cloud computing VMs have fewer I/O device requirements than general-purpose VMs. For example, Hyper-V offers up to 39 virtual devices to service its guests; these include a graphics card, a serial port, a DVD/CD-ROM, a mouse, a keyboard, and many other such devices that cloud VMs make no use of. In our experience, most cloud VMs only require processing, disk storage, and networking functionality from the underlying virtualization stack. Second, much of a virtual device's codebase is not required for handling the common cases of device use. For example, large swaths of code handle device initialization and power management, operations that are not critical to cloud guest VMs. In contrast, the code handling common device operations (e.g., reads and writes in case of a disk) is only a small piece of the overall virtual device's codebase. Finally, eliminating those virtual devices that emulate legacy hardware though low-level interfaces reduces complexity.

We present the design and implementation of Min-V, a cloud virtualization system based on Microsoft's Hyper-V. Min-V disables all virtual devices not critical to running VMs in the cloud and offers just nine virtual devices out of a set of 39. For the remaining devices, Min-V takes a step

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

further and virtualizes only their common functionality without handling device initialization or power management. To implement Min-V, we had to overcome a significant challenge: many commodity operating systems, such as generic configurations of Linux and Windows, fail to boot on our hypervisor. This occurs for two reasons: (1) the BIOS or the operating systems themselves check for the presence of several legacy devices at boot time, devices that are not critical to cloud VMs; and (2) they perform device initialization and power management, functionalities deliberately disabled in Min-V. One possibility is to rewrite commodity BIOSes and OSes. However, such an approach is challenging because it requires a significant amount of effort (numerous man-years of engineering) in a challenging development environment (low-level, pre-OS boot). Even worse, for cloud providers such an alternative may not be viable because they may lack source code access to the specific commodity OS a customer wants to boot in their environment.

As an alternative, Min-V sidesteps these engineering challenges with *delusional boot*. Delusional boot provides a full set of devices with complete functionality to the OS only during the boot process. Later on, it disables the non-critical devices and it replaces the critical ones with a set of barebones virtual devices that only offer common case functionality. With delusional boot, the guest VM is first booted using a normal configuration of Hyper-V with many of the virtual devices enabled, on a special node isolated from the rest of the datacenter. After the guest OS has finished booting, Min-V takes a VM snapshot, migrates it to a node in the datacenter, and restores the guest VM on the Min-V hypervisor that supports only the set of barebones virtual devices. Min-V uses a new TPM-based software attestation protocol that allows us to implement delusional boot using off-the-shelf servers and switches.

While delusional boot allows Min-V to successfully boot commodity OSes, guest OSes might still attempt to invoke functions via one of the removed devices. Min-V handles such accesses safely by returning a legitimate hypervisor error code (0xFF in Hyper-V). This simple error handling is sufficient to avoid crashing any of the commodity OSes (Windows 7, Windows XP, and Ubuntu) we tested. While these OSes panic during boot if they detect missing or faulty hardware, once running these OSes are often hardened against hardware errors that manifest gracefully through an error code. The graphics card best illustrates this behavior – although several commodity OSes refuse to boot if they lack a graphics card, they do not crash when the virtualized graphics hardware starts returning 0xFF at runtime. We used several workloads and a commercial reliability benchmark to investigate the robustness of delusional boot, and all three OSes tested remained stable throughout our experiments.

Of the 52 vulnerabilities described earlier, we estimate that removing virtual devices not needed in the cloud would eliminate at least 44 of them. Our estimate is conservative; for some of these vulnerabilities, it was not specified what device they belong to or whether they appear in the emulated portion of a device (such portions are removed by Min-V). Since Min-V also minimizes the functionality of the remaining devices, it is possible that even more vulnerabilities are eliminated; however, quantifying this effect on the data is much harder.

Our evaluation shows that Min-V reduces virtual device interfaces by 60% based on counting the number of lines source code in Hyper-V. This reduction in the attack surface is done with no performance penalty during the VM runtime. Min-V's sole performance overhead occurs when the guest VM needs to reboot, a relatively infrequent operation. In the case of Ubuntu, rebooting a guest VM in Min-V has an overhead of less than a minute in addition to the two minutes spent to boot the OS alone. We also evaluate Min-V's reliability using an industrial reliability benchmark and we found that all tested OSes remain stable.

2. Design Alternatives for Secure Cloud Virtualization

The need for secure virtualization for the cloud is greater than ever. Cases of cloud customers' software turning rogue have already been documented. For example, Amazon's EC2 machines have been used to send spam [21] and to launch denial-of-service attacks on other Amazon customers [3]. These examples demonstrate how cloud customers (knowingly or unknowingly) have started to abuse the cloud infrastructure for illegitimate activities. Compromised guest VMs can become launching pads for much more serious attacks on the cloud infrastructure or on other customers' VMs.

The security of cloud-based architectures rests on the virtualization stack and the hypervisor remaining uncompromised and enforcing isolation between guest VMs that may exhibit malicious behavior. One alternative is to start with a fresh design of a virtualization stack that uses small and well-defined paravirtualized channels for I/O communication. While such a clean-slate approach can offer strong security, it would also restrict the choice of the guest VM operating system to those that support this "special" virtual stack. Also, commodity OSes were designed to rely on a rich set of devices, and changing such an assumption requires serious re-engineering. Section 9 will present a more in-depth analysis of the requirements of porting a commodity OS to a minimal virtualization stack, and will describe why such an alternative is expensive. Instead, here we focus on design alternatives that continue to offer full virtualization stacks to guest VMs. Such approaches to secure cloud virtualization can be classified in three categories.

1. Commodity Hypervisors. Commodity hypervisors, such as Xen, VMware, and Hyper-V, can run commodity OSes with high performance. These systems can accommodate many guest VMs running simultaneously with adequate performance. Their codebases are continuously upgraded to

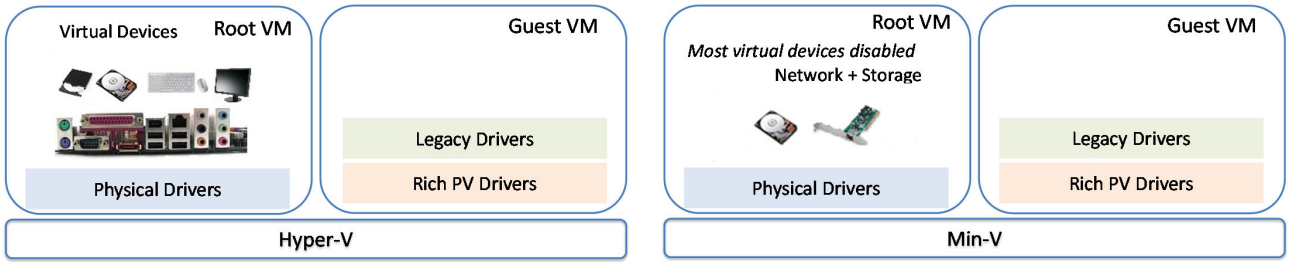

Figure 1. Hyper-V and Min-V Architectures: *In Hyper-V (on the left), the root VM offers a full set of virtual devices to the guest VMs. The guest VMs use legacy (hardware-emulation) drivers and rich paravirtualized drivers to interact with all virtual devices. In Min-V (on the right), the root VM disables most virtual devices. The only devices are left to support networking and storage. The guest VMs' drivers receive error codes if they try access non-existing devices.*

offer more functionality and higher performance. Unfortunately, these improvements come with a cost: the TCBs of these systems continue to grow. As the TCB grows, so does the potential for security vulnerabilities.

2. "Tiny" Hypervisors. Research projects have begun to investigate ways of offering virtualization without including a full-fledged operating system in their TCB [13, 22, 35]. For example, one recent project demonstrated a hypervisor with a very small codebase, only 7889 lines of C code [22], that is capable of running a commodity operating system. With such a small codebase, it may be possible to use verification to demonstrate that the system's implementation adheres to its specification [20]. However, to stay small, the hypervisor compromises on functionality and performance: while it is capable of virtualizing one commodity OS, it cannot run multiple guest VMs at the same time. Running multiple VMs simultaneously requires implementing multiplexing and demultiplexing functionality for I/O devices, which in turn requires much more code. For example, implementing a fast networking I/O path among co-located guest VMs requires code that implements the functionality of a network switch; in fact, Xen uses Linux's iptables and ebtables packages to implement switching. Such codebases tend to be large and complex.

3. Disaggregated Hypervisors. Another approach is to compartmentalize the virtualization stack in multiple isolated containers [8, 10, 24, 37, 42]. In some cases, the design of the disaggregation support is primarily driven by the reliability needs of the virtualization stack because each isolated container can be restarted upon a crash without needing a full system reboot [42]. Nevertheless, such isolation improves the overall system's security because an exploit within one container only compromises the data exposed within that container, rather than leading to a system-wide compromise [8, 37]. However, disaggregation only limits the impact of security vulnerabilities, it does not reduce the size of the TCB nor the number of vulnerabilities. Furthermore, delusional boot can be applied to disaggregated hypervisors as well. When such hypervisors need to boot a commodity OS that expects a full virtualization stack, delusional boot can simplify booting these OSes in a secure manner.

3. Threat Model

Cloud computing raises many different security threats and Min-V only addresses a portion of them. Our threat model assumes that attackers can run arbitrary code inside a guest VM with root privileges, and they have full access to the interfaces provided by the hypervisor. We make no assumptions about the configuration of the customer's VMs, because cloud providers prefer to impose minimal requirements on their customers. These OSes can be corrupt, they might not be upgraded or patched, and they may run rootkits and Trojans.

Min-V improves security on behalf of both cloud providers and cloud customers. Because cloud nodes often use the homogeneous software configurations, one exploit may compromise a large number of cloud nodes. Min-V protects cloud customers because, if an attacker compromises a cloud node's root VM, the attacker can further compromise other customers' VMs.

Cloud computing raises additional threats that are beyond the scope of our work. For example, a malicious administrator in a datacenter could try to compromise customers' guest VMs or steal physical disks loaded with customers' data. Alternatively, an administrator could be negligent in how they handle both software and hardware, which could lead to accidental data loss. Customers' data could also be subject to subpoenas [4], and data disclosures may not even be revealed to the customers [12]. Finally, software developers may introduce backdoors into the code which could be later exploited to gain access to guest VMs code and data.

4. Design Goals and Design Principles

This section provides a brief overview of the Min-V architecture, its design goals, and its design principles. Figure 1 shows an architecture diagram of our system and contrasts it with the original Hyper-V system.

4.1 Design Goals

1. Minimize the interface between the TCB and the guest VMs. To meet this goal, Min-V disables most virtual devices because they comprise most of the interface complexity between the TCB and the guest VMs.

2. Support legacy OSes in guest VMs. Min-V allows customers to run any legacy OS configurations and applications inside their guest VMs. In our experiments, we used three commodity OSes (Windows 7, Windows XP, and Ubuntu 9.10) which are representative of typical cloud environments. To narrow the virtual devices, Min-V replaces all remaining devices with a set of barebones virtual devices. This is done by installing a set of drivers in each OS. In addition of being smaller, these paravirtualized drivers offer fast performance.

3. Minimize the performance overhead. Performance is critical in cloud environments. Our goal is to meet the cloud provider's security needs without significantly impacting guest VM performance. To meet this goal, Min-V does not add any performance overhead to running VMs. As we will describe later, Min-V does increase the time it takes to reboot guest VMs.

4.2 Design Principles

In the context of the above design goals, four key principles guide our design:

1. Economy of interfaces. Any interface between the TCB of the virtualization system and the guest VMs that is not necessary for cloud computing should be eliminated.

2. Use high-level device interfaces rather than low-level ones. It is easier to secure high-level paravirtualized interfaces than to secure low-level legacy device interfaces.

3. Isolate a cloud node from the network whenever it executes potentially insecure operations. Whenever a guest OS must run with a full, commodity virtualization stack, it must be disconnected from the network to prevent compromises from spreading within the datacenter. The node must attest that it runs a minimal virtualization stack before being allowed to reconnect to the network.

4. Use little customization. Our solution should not require massive re-engineering of the OSes or special-purpose hardware, such as switches that incorporate trusted computing primitives into their logic. Such solutions are often expensive and they are hard to deploy in practice.

5. Disabling Virtual Devices

A virtualized cloud computing environment, such as Min-V, differs from a general purpose virtualization platform in that customers access guest VMs entirely via the network. As a result, many standard physical devices on cloud servers need not be exposed to the guest VM. For example, physical devices such as the keyboard, mouse, USB ports, and DVD drive serve little purpose for a cloud customer. In fact, many of these devices have virtual equivalents that are provided by remote desktop protocols such as RDP or VNC. For example, a customer running Windows in a guest VM can use RDP to redirect many devices over the network, such as the keyboard, mouse, graphics card, printer, and even USB storage.

In the rest of this section, we describe the types of devices that Hyper-V provides by default to guest VMs, and how they are powered-up and initialized. We then discuss the device requirements of operating systems that run in the guest VMs, and we describe the steps we took to actually remove devices from the virtualization stack.

5.1 Hyper-V Devices

Most *virtual devices* provided by Hyper-V to guest VMs correspond to real physical devices, such as the NIC, the IDE storage controller, or the keyboard controller. There are three common approaches to implementing a virtual device: 1) multiplexing the virtual device over the corresponding real physical device provided by the operating system running in the root VM; 2) emulating the hardware device entirely in software to provide the desired functionality; and 3) providing virtualization services through a device interface. As examples of the latter category, Hyper-V provides the *VMBus* device that provides a fast, shared-memory based communication channel between the root VM and a guest VM and a *heartbeat* integration component that provides a way to keep track of the guest VM's health status.

In its default configuration, Hyper-V offers 39 virtual devices to each guest VM. This large number of devices is not unique to Hyper-V; Xen offers a comparable number of devices to guest VMs. For each VM, Hyper-V creates and maintains a *virtual motherboard* device, which acts as a container for the set of internal devices available to that VM. Each virtual motherboard has a *virtual device manifest*, which is just a table that enumerates all devices found on the virtual motherboard. When a VM is initialized, each device listed in the manifest is instantiated and attached to the virtual motherboard. Once initialization completes, the motherboard and its devices are all powered on. At this point, virtual devices register their handlers with the hypervisor, so that guest VM accesses to certain I/O ports and MMIO addresses are dispatched to the appropriate virtual device.

Virtual devices often have dependencies on one another. For example, all enlightened devices depend on the VMBus device because the VMBus implements the shared memory bus between the root and a guest VM. Another example is the emulated NIC which depends on the emulated PCI bus to function properly. Figure 2 depicts the 39 devices found in Hyper-V as nodes which are connected by directed edges that represent dependencies. Because of these dependencies, the order in which devices are initialized is important. For example, the VMBus is the first device initialized by Hyper-V's virtual motherboard. Similarly, the emulated PCI bus is initialized before the emulated NIC. For Min-V, determining these device dependencies is important because *disabling a*

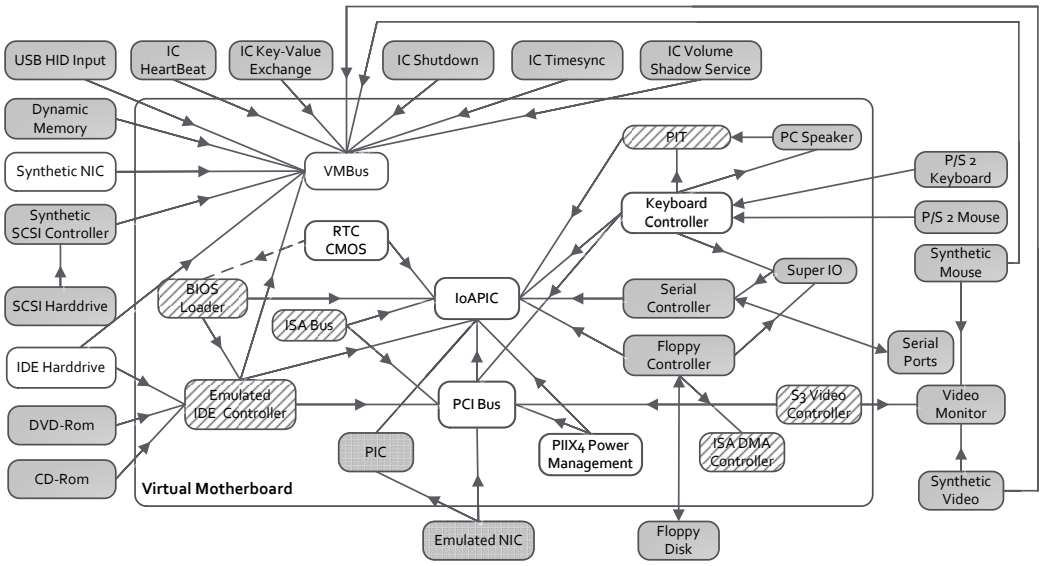

Figure 2. Virtual stack dependencies. *Devices that we simply removed from Min-V have a dark background. The stripe background represents devices that Windows 7 or the virtual BIOS check for at boot time, and we successfully "removed" them with delusional boot. Devices with a white background are those remaining in our system. Arrowed lines show virtual device dependencies, and the dashed line represents a dependency we manually removed.*

virtual device cannot be done unless all its dependent devices are also disabled. Next, we describe how we determined device dependencies.

5.2 Determining Device Dependencies

We use three heuristics to determine device dependencies in Hyper-V. Each heuristic allows us to test for the presence or absence of a dependency between two devices. While each heuristic has different shortcomings (e.g., some cannot find all dependencies, while others do not scale), their combination allows us to find all dependencies shown in Figure 2. Our heuristics cannot guarantee that *all* dependencies are discovered. However, any missed dependency represents a missed opportunity for further codebase reduction without affecting the performance or correctness of our current implementation.

1. Using object file references. We classify the object files created at compile time to determine which ones contain the functionality of a single device alone. Although in some cases an object file can include more than one device (e.g., the floppy disk and the floppy controller end up in the same object file after compilation), in most cases there was a one-to-one mapping between devices and object files. For such object files, we examine their symbol tables searching for external references to symbols defined in other object files. Whenever we find such a reference, we note a dependency between these two devices.

This heuristic is not guaranteed to find all dependencies because certain invocations may be performed as indirect calls though function pointers. It is challenging to use static analysis to identity such runtime dependencies, because you would need to locate each indirect jump instruction and then understand which instructions were used to calculate the jump target.

2. Disabling device instantiation. Another heuristic we use is to comment out devices in the virtual motherboard's manifest, one device at a time, and test whether a guest VM fails to boot. Such a failure indicates the presence of another device that depends on the removed device. We then automate the search for that other device by disabling different candidates until the OS boots successfully. This indicates a device dependency.

3. Code inspection. In some cases, we resort to code inspection to find additional dependencies. For example, we discovered a dependency between the emulated IDE controller and the VMBus which does not lead to OS instability. The IDE controller is a hybrid device that checks whether the VMBus is instantiated to take advantage of it for faster I/O. If the VMBus is not present, the IDE controller continues to function properly by falling back to emulation-only mode. While rigorous code inspection will find all dependencies between devices, this is challenging because of the codebase size and the error-prone nature of the process. Instead of relying only on code inspection, we use the first two heuristics to quickly discover many device dependencies, and we only use code inspection to supplement their shortcomings.

Based on the combination of these heuristics, we discovered 68 device dependencies. Figure 2 uses arrows to illustrate all dependencies except for those involving the virtual motherboard. There are 17 devices dependent on the virtual

motherboard and they are all shown within a bounding box representing the virtual motherboard. Heuristic #1 discovered 55 device dependencies including all the 17 dependencies involving the virtual motherboard. Heuristic #2 discovered 11 device dependencies, and the remaining two dependencies were discovered using the last heuristic.

5.3 Removing Devices

Because Min-V targets cloud virtualization environments, our goal is to provide the minimum set of devices needed for guest VMs running in the cloud: a CPU, a clock, an interrupt controller, a disk, and a NIC. After discovering device dependencies, we naively thought that we could disable all devices except for the virtual motherboard, the RTC CMOS, the IoAPIC, the enlightened NIC, the IDE harddrive, and their dependency, which is the VMBus. However, we quickly ran into three obstacles.

First, the Hyper-V boot model requires the presence of the IDE harddrive device and its dependencies (the emulated IDE controller, and the PCI Bus). Hyper-V uses a boot model for guest VMs that mirrors the way in which physical machines boot. In particular, Hyper-V provides a virtual BIOS that expects the OS to boot from a legacy virtual IDE. To overcome this obstacle, the BIOS must be paravirtualized in a way that allows the guest OS to boot over the VMBus.

Second, the Hyper-V BIOS requires the presence of five devices. They are the BIOS loader, the RTC CMOS, and the keyboard, video, and ISA DMA controllers. Unfortunately changing the BIOS to eliminate these dependencies was impossible because we did not have the source code of the BIOS. Even with access to the BIOS source code, we expect modifying it to be a challenging task because of the low-level, pre-OS nature of the environment.

Third, commodity operating systems often check for the presence of certain devices at boot time in order to initialize them. For example, Windows 7 and Ubuntu 9.10 check for the presence of a video card and panic if they do not find one. To identify which devices are expected at boot time, we ran a series of experiments where we would disable one device at a time, attempt to boot an operating system, and check if the OS would panic. To perform these experiments, we began by disabling the devices which had no dependencies, and we made sure never to disable a device where any its dependencies were not already disabled. We used three different OSes; the results are presented in Table 2.

In summary, one way to overcome these three obstacles is to add the VMBus driver support to the BIOS, rewrite the BIOS to remove its dependencies on devices which are not needed for the cloud, and paravirtualize all the guest OSes to no longer attempt to initialize devices at boot time other than the synthetic NIC and disk. However, such a plan requires a drastic amount of effort, comparable to implementing and testing a new release of a commodity operating system.

Min-V overcomes these challenges in two steps.

Virtualized BIOS	BIOS loader, Keyboard controller, Video, ISA DMA Controller, RTC CMOS
Windows 7	PIT, ISA Bus, Video, RTC CMOS, Power management
Windows XP	PIT, ISA Bus, Keyboard controller, RTC CMOS, Power management
Ubuntu 9.10	PIT, Video, RTC CMOS, Power management

Table 2. Devices Needed at Boot Time.

5.3.1 Step #1: Removing extraneous devices

The first step is removing extraneous devices – devices whose removal does not raise any of the challenges shown above. For this, we modify the virtual device manifest to remove all the extraneous devices at the time a guest VM is created. These devices will neither be initialized nor powered on by Hyper-V.

5.3.2 Step #2: Using delusional boot

In step # 2, Min-V uses a technique called *delusional boot*: the guest VM is first booted, using a normal configuration of Hyper-V with many virtual devices enabled, on a special node that is isolated from the rest of the datacenter. After the guest OS finishes booting, Min-V takes a snapshot by pausing the VM and saving its state. Min-V then migrates the VM snapshot to the datacenter production environment, and restores the guest VM using a version of Hyper-V that only offers a barebones set of virtual devices. Section 6 will provide an in-depth discussion of the delusional boot technique. Figure 3 illustrates the devices left in the Min-V virtualization stack after removing 24 devices in step #1 and removing an additional six virtual devices in step #2. Both steps are critical to reducing the virtualization stack's codebase; we leave an in-depth examination of how much code each step eliminated to the evaluation section.

One challenge that arises with delusional boot is *safety*: what happens if the guest OS attempts to access one of the devices that was removed after boot? If such accesses are not handled or prevented, they could potentially lead to guest OS instabilities. Min-V uses two techniques to address this challenge. First, some devices can be safely removed before booting the OS. These devices include certain Hyper-V services, such as the VM heartbeat device, plug-and-play devices, and also physical devices often missing from many PC configurations, such as a floppy drive. By disabling these devices, the commodity OS never learns about their existence. Second, Min-V uses remote desktop protocols (e.g., RDP and VNC) that virtualize several missing devices, such as the keyboard, mouse, and display. All I/O to and from these devices is redirected over the network interface, an interface still available in Min-V. To increase confidence in our examination, we performed a series of reliability experiments whose results are described in our evaluation section.

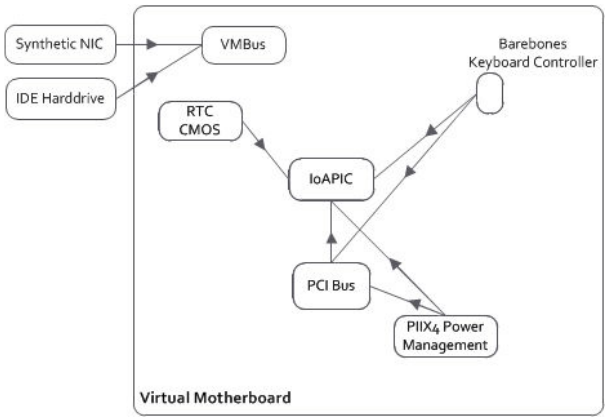

Figure 3. The virtual devices left in Min-V's virtualization stack. *Nine devices are left in Min-V, out of which three are paravirtualized and five are emulated. While the set of paravirtualized devices and the set of emulated devices appear disconnected in the illustration, both sets are instantiated through the virtual motherboard device.*

5.3.3 Implementation Details

We modified the Hyper-V virtual motherboard initialization code to skip all devices not needed at boot time. These devices will not register their handlers with the hypervisor, and guest accesses to their I/O address spaces fall back to the *null* device model: all writes are discarded, and all reads returns a default error value (for Hyper-V, this value is 0xFF). We also eliminated a dependency between the clock (RTC CMOS) and the BIOS loader. At boot time, the RTC code initializes itself by calling into the BIOS to retrieve the clock. Since this code is never executed after the OS has booted, we simply rewrote the RTC CMOS device in Min-V to eliminate this dependency.

To implement a working CPU, a NIC, a disk, an interrupt controller, and a clock, Min-V must offer six devices: the virtual motherboard, the VMBus, the synthetic NIC, the synthetic IDE harddrive, the RTC CMOS, and the IoAPIC (which the clock depends on). However, our current implementation ends up offering three additional devices: the PIIX4 power management device, the PCI bus, and the keyboard controller. Hyper-V uses PIIX4 to save/restore VMs, a functionality Min-V also requires for delusional boot. The PIIX4's role is to dispatch hibernate commands to the guest OS so that the OS can save its state appropriately before shutting down. Because the PIIX4 device uses the PCI bus, Min-V also has to offer the PCI bus device. Finally, Min-V needs the keyboard controller (not to be confused with the keyboard device) because guest VMs use it to reset their CPUs.

We took additional steps to further minimize the codebase of the remaining devices. For example, the keyboard controller device offers four pieces of functionality out of which only one is needed by Min-V. These are: (1) controlling and receiving inputs from the PS/2 keyboard and mouse; (2) controlling the A20 line [36] to enable protected mode; (3) controlling the PC speaker; and (4) controlling the guest VMs' CPU reset. We rewrote the keyboard controller device to remove the first three pieces of functionality which are not needed by Min-V. Figure 4 illustrates the pseudo-code of the original device (on the left) and the newer, barebones, keyboard controller used by Min-V (on the right). Although we only re-implemented the keyboard controller, reducing all remaining virtual devices to barebones devices is left as future work.

```
PowerOn //Before delusional boot

- get reference to SuperIO device
- get reference to PCIBus device
- get reference to IOApic device
- get reference to PIT device
- get reference to Speaker device

//setup PS/2, A20, speaker, CPU reset
- setup IO emulation for port 0x60
- setup IO emulation for port 0x61
- setup IO emulation for port 0x62
- setup IO emulation for port 0x63
- setup IO emulation for port 0x64

- initialize PS2 keyboard
- initialize PS2 mouse
```

```
PowerOn //After delusional boot

- get reference to PCIBus device
- get reference to IOApic device

//setup CPU reset only
- setup IO emulation for port 0x64
```

Figure 4. Re-implementing the keyboard controller. *After the delusional boot, we can switch to a barebones keyboard controller device.*

6. Delusional Boot

At a high-level, there are three steps to delusional boot: 1. copying the customer's VM image to an isolated boot server, 2. booting the VM on the isolated server and 3. copying the VM image back to the production environment.

1. Copying the VM Image to the Boot Server. Min-V detects that a guest VM is ready to be rebooted by interposing on the keyboard controller port 0x64. This port links to the CPU reset pin in the original x86 PC architecture and tells the hypervisor that the guest OS has finished shutting down and the VMM stack should be rebooted. At this point, Min-V saves the guest VM to a file and requests service from an available boot server.

2. Isolated Boot. Delusional boot relies on an *isolated* boot server located in an environment separate from the rest of the production nodes (e.g., a *quarantined* environment). This boot server runs the *Min-V boot stack*, a different version of the virtualization stack than the *Min-V production stack* that runs in production. First, the isolated boot server copies the guest VM disk image and VM configuration from the production server, disconnects from the network, and then reboots into a configuration that offers a full virtualized stack. At this point, the server boots up the VM. After the guest OS finishes booting on the boot server, the role of the full

virtualization stack is now complete. The boot server then *snapshots* the VM state (including the virtual device state) to a file, and then reboots and reconnects to the network.

3. Copying the VM image back to Production. The VM snapshot is migrated back to a production server, and the Min-V virtualization stack on this server replaces all the disabled virtual devices with a *null device* virtualization model. This model treats all accesses to these devices as no-ops. In particular, all memory-mapped and port-mapped device reads return the value 0xFF, and all writes are discarded. Together, these steps complete the delusional boot and achieve the end goal of running a guest VM on the production server with enhanced isolation properties.

6.1 Threat Model of Delusional Boot

We designed delusional boot to handle the following three classes of attacks:

1. Vulnerabilities that persist across reboots. Such an attack could be launched by booting a malicious VM which would then install a rootkit (e.g., through a firmware exploit) on the boot server. If left unhandled, an installed rootkit can compromise all future guest VMs booting on the server. To stop such attacks, our implementation of delusional boot uses a TPM to measure all firmware and the entire OS image booting on the server. If any modifications are detected in this software, the boot process stops because new injected code has been discovered on the server's boot path. While such an approach stops injected code from surviving reboots, it does not eliminate the vulnerability. Fixing such a vulnerability requires an OS or firmware update.

2. Exploiting a bug in the VM migration protocol. Numerous optimizations are possible to perform fast VM migration (e.g., compression, deduplication), and many such optimizations are implemented in commodity virtualization solutions. However, such optimizations require running a larger or more complex software stack on the boot server. To reduce the possibility of an exploit in the VM migration protocol, our implementation of delusional boot is deliberately kept simple – it is just a networking transfer of the VM image.

3. The VM of one customer infecting the VM of another. The boot server offers a full virtualization stack to a guest VM for rebooting purposes. A guest VM can exploit a vulnerability and compromise a co-located VM. To eliminate this possibility, we require the boot server to run co-located VMs only when they belong to the same customer.

6.2 Securing Delusional Boot

A guest VM may try to compromise the boot server's virtualization stack because it exposes a large interface consisting of a full set of virtual devices. However, our design relies on two properties to ensure such compromises do not lead to security breaches. First, the boot server's network connection is disabled unless it is able to attest (using a TPM) to running a pre-established, sanitized software configuration; such a configuration *never* runs third-party guest VMs. Second, only guest VM snapshots are loaded back from the boot server into production, and they are subjected to the same security protocol used when importing any untrusted customer VM image. This ensures any compromises remain isolated inside the guest VM.

6.2.1 Modes for the Isolated Boot Server

The isolated boot server offers two modes of operation: 1) a *clean mode* only used when importing and exporting customer VM images in and out of the isolated boot environment; and 2) a *dirty mode* used for actually booting the customer-provided VMs. The network switch connecting the isolated boot server moved is configured to only offer network connectivity when the boot server is in clean mode.

Initially, we planned to use a TPM-based attestation protocol [32] to detect the boot server's configuration (clean vs. dirty). The boot server would produce a TPM-signed software attestation which would be transmitted to the switch. The switch would verify the attestation before enabling access to the network. There is already an open protocol designed to solve this exact problem, namely TCG's Trusted Network Connect (TNC). Although switch manufacturers are starting to adopt TNC, we could not find an inexpensive, commodity switch that supports TNC. We overcame this temporary obstacle by designing a new software attestation protocol that works with commodity switches. We only require the switch to support IP and MAC address filtering for access control, which is widely available today. In addition to IP and MAC filtering, our software verification protocol uses the boot server's TPM chip and Microsoft's BitLocker.

6.2.2 Min-V Software Attestation Protocol

We start by configuring the switch with a whitelist of MAC and private IP addresses; the switch enables network connectivity whenever a boot server's NIC presents one of the whitelisted addresses. For any other addresses, the switch denies network connectivity. The software attestation protocol ensures that a boot server can configure the NIC with valid IP and MAC addresses only when booted in clean mode. If the boot server is booted in dirty mode, our protocol ensures that the server cannot configure valid IP and MAC addresses.

To explain how our protocol works, we start with a quick overview of BitLocker, a disk volume encryption feature of Windows that uses the TPM to protect its volume encryption keys. BitLocker can only retrieve the encryption key if the following two conditions hold. First, decryption must be done on the same machine that encrypted the volume. Second, the machine's boot configuration, as recorded by the TPM, must match the configuration that saved the volume

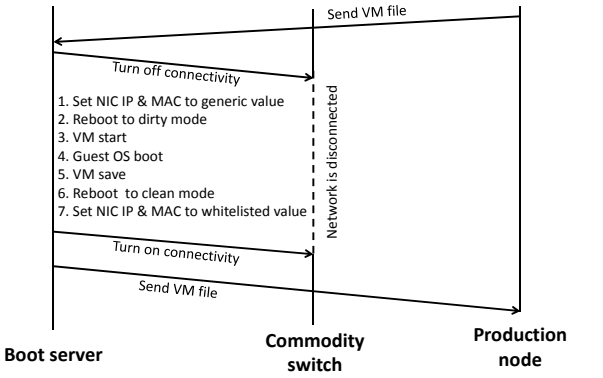

Figure 5. The software attestation protocol in Min-V.

encryption key. To provide these two guarantees, BitLocker *seals* the the encryption key to the chain of trust rooted in the physical TPM and recorded in the TPM's Platform Configuration Registers (PCRs). The PCR values consist of hashes of the BIOS, the I/O devices' firmware, the Master Boot Record (MBR), the Windows Boot Manager (BootMgr), and the boot configuration data. The key can only be *unsealed* by the boot manager *before* the OS is launched while the PCR registers are set to the appropriate values. The key remains safe because an attacker cannot modify the boot manager or else the PCR values will not match and the unseal operation will fail.

Min-V uses a modified version of BitLocker that protects the boot server's whitelisted IP and MAC addresses in the same way that BitLocker protects its volume encryption keys. The valid addresses are unsealed successfully only if the boot server has booted in clean mode. Any other configuration (i.e., dirty mode) cannot unseal the valid addresses, and without these addresses the network traffic will be blocked by the switch. Guessing the "correct" MAC and IP addresses is hard; the search space is 72 bits long. Also, the network switch is configured to isolate each boot server from any other device to prevent network sniffing attacks. Figure 5 shows the Min-V software attestation protocol.

6.3 Security Discussion

If a boot server becomes compromised, it might try to: 1) attack other nodes in the cloud infrastructure; 2) compromise the clean mode execution environment on the boot server; or 3) launch a DoS attack by refusing to restart the boot server in clean mode. We consider each of these attacks in turn.

To prevent the first class of attacks, Min-V ensures that network connectivity to the production nodes is disabled in all configurations other than the clean one. The network switch will not re-enable the network port unless the boot server can configure its NIC with the valid IP and MAC addresses. The boot server cannot retrieve these addresses hidden in the encrypted volume unless it is booted in a clean configuration. Any other configuration wanting to decrypt the partition storing these addresses would need to modify the boot manager to unseal the key protected by BitLocker. Modifying the boot manager leads to a mismatch in the PCR values, which prevents the unseal operation from revealing the key.

To prevent the second attack, the trusted clean execution environment is also stored on a TPM-sealed partition. This prevents a malicious boot server from modifying the encrypted partition where the trusted execution environment is stored. While the boot server could potentially delete the entire encrypted partition and put a new malicious version in its place, this would simply delete the whitelisted addresses, and prevent the boot server from successfully attesting to the network switch. When the boot server runs in clean mode, the customer's VM image is only stored as a file and is never activated.

Finally, Min-V does not currently prevent the third class of attack. When booted in dirty mode, a compromised boot server might refuse to reboot back into clean mode, effectively mounting a DoS attack. Another way to mount a DoS attack would be for the compromised boot server to modify or delete the clean configuration. For example, it could delete the encrypted volume or manipulate the boot manager's configuration parameters for booting in clean mode. This would cause clean mode to no longer boot successfully since the TPM unseal operation would fail. In the future, Min-V could mitigate certain DoS attacks by using an out-of-band control mechanism, such as Intel's management processor [15], which can force a boot server to reboot into clean mode.

6.4 Performance Discussion

Two performance concerns with delusional boot are that (1) it introduces additional latency overhead to guest VM rebooting and (2) the boot server can become a bottleneck if multiple VMs need to reboot simultaneously. As Section 7 will show guest VM reboots are already relatively slow spanning multiple minutes and the additional overhead due to delusional boot is small. To alleviate the second performance concern, Min-V can rely on a small cluster of boot servers rather than just on a single machine. Such a design should easily scale because there is no state sharing across these boot servers. We also examined a trace of server reboots collected across 130 production nodes over the course of one month. Server reboots in the cloud were relatively rare (only 10 servers rebooted) and there were only two occurrences of simultaneous reboots. In both cases, two servers rebooted simultaneously. Finally, it is possible for a single boot server to reboot multiple guest VMs *as long as* they belong to the same customer since Min-V does not need to isolate guest VMs belonging to the same customer. While such a design extension is possible, we leave it as future work.

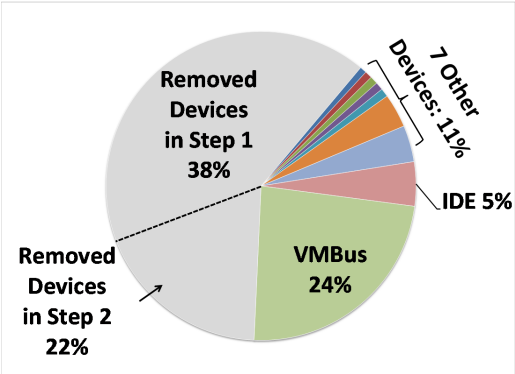

Figure 6. Percentage of lines of virtual devices code removed by Min-V.

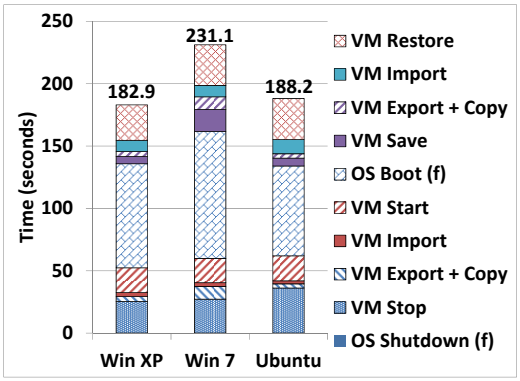

Figure 7. Performance of Delusional Boot. *The overhead of delusional boot is broken down for each of the steps taken by the protocol. There is a total of ten steps. Only two steps (OS Shutdown and OS Boot) appear in a regular boot, and thus we marked them with "(f)" (standing for "fixed cost"). The remaining eight steps are all overhead due to delusional boot.*

7. Evaluation

This section presents a three-pronged evaluation of Min-V. First, we measure how much smaller the virtual device interface is for Min-V. Second, we evaluate the performance of delusional boot by measuring its operational latency. Finally, we evaluate the stability of guest OSes in Min-V.

Methodology In our experiments with Min-V we used Intel Core 2 Duo machines, each equipped with a E6600 2.4GHz CPU and 3GB of RAM. These machines were linked via 1Gbps dedicated network. We experimented with three commodity operating systems: Windows XP Professional, Windows 7 Ultimate, and Ubuntu Linux version 9.10 (kernel ver. 2.6.28-16 SMP x86_64). Each commodity system ran inside of a guest VM that was allocated 1GB of RAM and a dynamically-sized virtual disk set to a maximum of 120GB. The performance experiments were obtained by repeating each experiment three times and reporting the average results. There is very little variance across the different runs of our experiments.

In our delusional boot experiments, we used the Hyper-V's differential VHDs mechanism to minimize the amount of data shipped between the production nodes and the boot server. To implement this, we assumed the boot server already has saved the original "golden image" of the VHDs for each commodity OS.

In our evaluation of boot server throughput, we used a trace of node reboots in the cloud. Our trace comes from a live cluster of 130 servers as part of *anonymized* application running in a large datacenter; the trace spans 1.5 months from mid January 2009 to early March 2009.

7.1 Reducing the Attack Surface

We use the number of lines of source code to evaluate our reduction in the virtual device interface. While the relationship between code size and number of vulnerabilities is not precisely known, software engineers estimate that the density of bugs in production quality source code is about one to ten bugs in 1000 lines of code [14]. The codebase implementing the virtual device interface in Hyper-V is on the order of a few hundred thousand lines of code. Figure 6 shows the percentage of the lines of code that Min-V eliminates from the interface to the TCB by removing virtual devices. To reduce the clutter in the graph, we collapsed the names of seven devices we remove into one label: "7 Other Devices". These seven devices constitute 11% of the codebase and they are: the NIC (4%), virtual motherboard (4%), and the PIIX4 power management, the PCI bus, the keyboard controller, the I/O APIC, and the RTC CMOS, each with 1% or less.

Both device removal steps are effective in reducing our codebase size. Removing extraneous devices (step #1) provides a 38% codebase reduction, whereas delusional boot (step #2) gives an additional 22% reduction in the codebase. The VMBus constitutes the bulk of the remaining codebase. This is encouraging because the interfaces exposed by the VMBus are based on high-level message passing rather than the memory-mapped IO of legacy devices. Our experience with devices tells us that new, paravirtualized devices are much more secure (because of their style of interfaces) than legacy ones. Furthermore, we have started to build *VMBus light*, a smaller version of the VMBus that removes some of the functionality unnecessary in the cloud, such as power management and initialization.

7.2 Delusional Boot Latency

We instrumented our implementation of delusional boot to record time spent in each of its steps required to reboot a customer's guest VM.

Figure 7 shows the performance of delusional boot for the three commodity OSes we used. The OS shutdown and boot steps are the only two steps present in a regular boot,

whereas the rest of the steps is the overhead introduced by delusional boot. The end-to-end cost of booting an OS in Min-V is 3.05 minutes to complete for Windows XP Professional, 3.85 minutes for Windows 7 Ultimate, and 3.14 minutes for Ubuntu. In contrast, the cost of regular reboot in Hyper-V is 2.16 minutes for Windows XP, 2.83 minutes for Windows 7, and 2.2 minutes for Ubuntu. The difference in the performance of Windows 7 on one side and Windows XP and Ubuntu on the other is due to the different image sizes of these OSes. The size of a fresh Windows 7 install is 8.5GB as opposed to 3.4GB for Windows XP and 3GB for Ubuntu. A larger image size increases three steps in the delusional boot process: the VM export + copy, the OS boot, and the VM save.

7.3 Reliability

With Min-V, guest OSes operate in an environment where many virtual devices are either missing or not properly emulated. This could introduce reliability issues if the OS would want to access a virtual device and this access wouldn't be properly handled by Min-V. To evaluate whether Min-V remains reliable, we used an industrial reliability benchmark called PassMark BurnIn Test. This benchmark tests the I/O subsystems of a traditional PC for reliability and stability; one common use of this benchmark is to test whether PCs remain stable when their CPUs are overclocked. This experiment's goal is to detect whether the OS would crash given the large load of I/O calls made to the virtualization stack. A run of the benchmark completed in about 15 minutes on the machines used in our evaluation. Figure 8 shows a summary of the test results from running PassMark on Windows 7 in Min-V.

The main finding of our experiment is that all three OSes remained stable even when booted with delusional boot. We closely inspected the logs and compared them to the logs obtained when running the benchmark on an OS that booted normally. We found the logs to be similar overall: many tests passed on both normally booted and delusional booted OSes while some tests failed. Most of the failed tests showed identical errors. However, this was not always the case. For example, one test checked whether the graphics card supports hardware-only DirectDraw object creation operations. This test succeeded when we booted the OS normally because the virtual device is able to relay DirectDraw commands. Min-V uses RDP and the RDP graphics driver does not support DirectDraw. This check however made the benchmark conduct an additional series of tests for DirectDraw on the normally booted OS, and many of these tests failed. In fact this causes the error count to be higher for the normally booted OS than the delusional booted one.

To increase our confidence in the reliability of our guest VMs, we also installed and ran four common applications: a browser, a Web server, an FTP server, and an SSH server. We used these applications ourselves over the course of two days (48 hours), and we experienced no instabilities. In all our reliability tests whether done through the benchmark or whether by running common applications, access to removed devices were handled safely, and all three OSes remained stable.

Test Name	Cycles	Operations	Result	Errors
CPU	165	78.231 Billion	PASS	0
Memory (RAM)	3	4.044 Billion	PASS	0
Printer	1	5714	PASS	0
Network	16	135600	PASS	0
Video (RDP)	12	376	PASS	0
Disk (C:)	2	7.024 Billion	PASS	0
2D (Graphics)	0	0	FAIL	85
3D (Graphics)	0	0	FAIL	5
Sound	4	7.96 Million	FAIL	260
Parallel Port	0	0	FAIL	85
Tape	0	0	FAIL	57
USB Plug 1	0	0	FAIL	1
USB Plug 2	0	0	FAIL	1
Serial Port 1	0	0	FAIL	78
Serial Port 2	57	0	FAIL	58
Disk (A:)	0	0	FAIL	3

Figure 8. Summary of results from running PassMark on Windows 7 in Min-V.

8. Related Work

The most common approach to improving the security of hypervisor-based systems is to reduce the size of the TCB [13, 19, 22, 26, 35]. For example, SecVisor [35] provides kernel code integrity using a very small hypervisor combined with hardware support for memory protection. TrustVisor [22] enables efficient data protection and execution integrity using a small special-purpose hypervisor, yet it only supports one guest VM, and therefore is not suitable for cloud computing environments. NoHype [19] goes one step further to entirely remove the virtualization layer, yet it requires additional hardware support and it only provides static partitioning to allocate machine resources. NOVA [37] relocates non-critical components into user mode to reduce its TCB, but device drivers need to be rewritten from scratch. In contrast, Min-V's goal is not to reduce the TCB's size, but rather the size of the interface between the TCB and the guest VMs. Min-V assumes as little as possible regarding the guest OS.

Min-V is not the first system to investigate narrowing the interfaces to make virtualized systems more secure. Bunker [23] uses a crippled OS with restricted I/O drivers to implement secure network trace analysis code, and Terra [11] uses tamper-resistant hardware to offer a closed-box abstraction on commodity hardware. Min-V differs from these systems primarily in the target application: virtualized cloud computing imposes different device requirements and requires support for commodity OSes.

Another approach to improving the security of virtualized systems consists of disaggregating the TCB by partitioning it into isolated components [8, 9, 10, 24, 37]. This approach does not eliminate vulnerabilities from the TCB, but instead it limits their impact. This isolation improves security because an exploit in one container only compromises the data exposed to that container. A recent project [8] takes a step further and uses microreboots to restart some of the hypervisor's components in an effort to reduce their temporal attack surface. Such techniques are different than Min-V's delusional boot whose goal is to support booting commodity OSes without re-engineering the startup code.

Cloud security is becoming a significant concern, and several research efforts have proposed solutions to address: storing data in the cloud [1, 5], nested virtualization [2, 18], side-channel attacks in cloud infrastructure [28], preventing information leakage for map-reduce applications [29], information flow-control between VMs [31], and enabling confidentiality and integrity of customer computations in the cloud [16, 33, 43]. Unlike Min-V, none of these efforts focus on reducing the impact of security vulnerabilities in the virtualized systems' TCB.

Device drivers are a well-known source of reliability and security issues for OSes [7, 38]. Much of the research effort in this area has focused on minimizing the impact of driver bugs [17, 30, 39] to make operating systems more reliable. However, one recent effort [6, 41] moves device drivers out of the TCB by running them in user-space. This approach requires significant modifications to device drivers and conflicts with our design principle of using little customization.

9. Discussion

9.1 Porting Commodity OSes to a Minimal Virtualization Stack

One alternative to delusional boot is a clean-slate approach – porting all OSes to eliminate their dependencies on legacy virtual devices. Such an approach could start by defining a new set of interfaces between the hypervisor and the guest environments. Such interfaces do not need to emulate legacy hardware. Thus, they can be much simpler and offer narrow APIs than today's interfaces based on hardware emulation. Such a clean-slate approach is attractive because it can potentially offer security and robustness guarantees beyond those of Min-V. However, such a porting effort needs to overcome two challenges.

The first challenge is maintaining backward compatibility. Even today, several Linux configurations and Windows Embedded can run "headless" and have fewer device dependencies than those of commodity Linux and Windows versions. However, such configurations often offer more limited functionality (e.g., fewer libraries, less flexibility) than their commodity counterparts. We believe cloud providers strongly desire running commodity OSes in their datacenters, ideally with no change from their non-virtualized counterparts. Such backward compatibility would allow customers to image their physical machines in order to migrate them to the cloud. An effort-less cloud migration story would make the cloud computing vision very compelling.

Second, porting the OS is insufficient. Today, all configurations of VMware and Hyper-V, and most configurations of Xen boot guest VMs using a legacy BIOS interface. Only after boot, the OS kernel can switch to a paravirtualized interface. Even an OS designed to run on a minimal paravirtualized interface will continue to use legacy devices at boot time. Delusional boot could offer an inexpensive way of bypassing the legacy BIOS dependencies and not include them in the TCB.

9.2 Evaluating the Complexity of the Virtualization Stack

Min-V's goal is to make the virtualization stack "simpler" in order to reduce the hypervisor's attack surface. However, it is unclear what the "right" design is to build a "simpler" interface. Our evaluation section used lines of code as a metric, but such a metrics is far from perfect. In our experience, it is very easy to introduce bugs when writing devices that emulate hardware because hardware APIs are arcane and complex.

For example, the codebase of a keyboard controller device is relatively small. Yet, the code implementing such a device in software is quite minute and elaborate. A keyboard controller offers a register-based interface. Often, the bits of these registers have different meanings depending on the device's manufacturer or the context in which the device is used. The code needs to handle all these special cases correctly, making heavy use of pointers, shifts, and XORs. We found it quite easy to make mistakes when writing such low-level code.

10. Conclusions

This paper presents Min-V, a system that improves the security of commodity virtualization systems. By disabling all virtual devices not critical to running customer VMs in the cloud, Min-V minimizes the codebase of the virtualization stack. To accomplish this without significant re-engineering of the guest operating system, we introduce delusional boot. Delusional boot ensures that commodity OSes can boot in the absence of many legacy devices that Min-V eliminates. Our evaluation shows that Min-V's security improvements incur only a small performance overhead during boot time. Our reliability tests show that Min-V is able to run unmodified Windows and Linux OSes on top of this minimal virtualization interface.

Acknowledgments

This paper benefited from comments and insights provided by Timothy Roscoe (our shepherd), Paul England, Brandon

Baker, Andy Warfield, Sam King, Krishna Gummadi, and the anonymous reviewers. We are grateful for their help.

References

[1] G. Ateniese, S. Kamara, and J. Katz. Proofs of Storage from Homomorphic Identification Protocols. In *Proc. of the 15th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2009.

[2] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *Proc. of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[3] BitBucket. On our extended downtime, Amazon and whats coming, 2009. `http://blog.bitbucket.org/2009/10/04/on-our-extended-downtime-amazon\discretionary{-}{}{}and-whats-coming/`.

[4] Boston Globe. Google subpoena roils the web, January, 2006. `http://boston.com/news/nation/articles/2006/01/21/google_subpoen\a_roils_the_web/`.

[5] K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[6] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *Proc. of the 2010 USENIX conference (ATC)*, 2010.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[8] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Cooker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Funtionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, 2004.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. of the 1st Workshop On Operating System and Architectural Support for the on demand IT Infrastructure (OASIS)*, Boston, MA, October 2004.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[12] R. Gellman. Privacy in the Clouds: Risks ot Privacy and Confidentiality from Cloud Computing, 2009. `http://www.worldprivacyforum.org/pdf/WPF_Cloud_Privacy_Report.pdf`.

[13] M. Hohmuth, M. Peter, H. Hartig, and J. S. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual-machine monitors. In *Proc. of 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[14] G. J. Holzmann. The logic of bugs. In *Proc. of Foundations of Software Engineering (FSE)*, Charleston, SC, 2002.

[15] Intel. Intel Active Management Technology. `http://www.intel.com/technology/platform-technology/intel-amt/`.

[16] M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono. On technical security issues in cloud computing. In *Proc. of the IEEE International Conference on Cloud Computing (CLOUD-II)*, Bangalore, India, 2009.

[17] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proc. of the 22nd Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[18] B. Kauer, P. Verissimo, and A. Bessani. Recursive virtual machines for advanced security mechanisms. In *Proc. of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV)*, 2011.

[19] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proc. of 37th International Symposium on Computer Architecture (ISCA)*, Saint-Malo, France, 2010.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the 22nd Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[21] B. Krebs. Amazon: Hey Spammers, Get Off My Cloud. Washington Post, July 1 2008.

[22] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[23] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown. Bunker: A Privacy-Oriented Platform for Network Tracing. In *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2009.

[24] D. Murray, G. Miob, and S. Hand. Improving Xen Security Through Disaggregation. In *Proc. of the 4th ACM International Conference on Virtual Execution Environments (VEE)*, Seattle, WA, March 2008.

[25] National Institute of Standards and Techonology. National Vulnerability Database. `http://nvd.nist.gov/home.cfm`.

[26] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proc. of the 2009 Annual Computer Security Applications Conference (ACSAC)*, Honolulu, HI, 2009.

[27] S. Özkan. CVE Details: The ultimate security vulnerability datasource. `http://www.cvedetails.com/index.php`.

[28] 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 *Proc. of 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, November 2009.

[29] I. Roy, H. E. Ramadan, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *Proc. of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2010.

[30] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. of the 4th ACM European Conference on Computer Systems (Eurosys)*, Nuremberg, Germany, 2009.

[31] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC 23511, IBM Research, 2005.

[32] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of the 13th USENIX Security Symposium*, San Diego, CA, 2004.

[33] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, San Diego, CA, June 2009.

[34] Secunia. Secunia Advisories. `http://secunia.com/advisories/`.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[36] T. Shanley. *Protected mode software architecture*. Taylor & Francis, 1996.

[37] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, Paris, France, April 2010.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. of the 19th Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003.

[40] VMware. Security Advisories & Certifications. http://www.vmware.com/security/advisories/.

[41] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proc. of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2008.

[42] Xen. Xen User Manual v3.3. http://bits.xensource.com/Xen/docs/user.pdf.

[43] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.