BitVisor: A Thin Hypervisor for Enforcing I/O Device Security

Takahiro Shinagawa
University of Tsukuba
shina@cs.tsukuba.ac.jp

Hideki Eiraku
Kouichi Tanimoto
University of Tsukuba
{hdk,tanimoto} @osss.cs.tsukuba.ac.jp

Kazumasa Omote
Japan Advanced Institute of Science and Technology
omote@jaist.ac.jp

Shoichi Hasegawa
Takashi Horie
University of Tsukuba
{s-hase,horietk} @osss.cs.tsukuba.ac.jp

Manabu Hirano
Toyota National College of Technology
manabu@dsl.gr.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@ci.kyutech.ac.jp

Yoshihiro Oyama
University of Electro-Communications
oyama@cs.uec.ac.jp

Eiji Kawai
Nara Institute of Science and Technology
eiji-ka@is.naist.jp

Kenji Kono
Keio University
kono@ics.keio.ac.jp

Shigeru Chiba
Tokyo Institute of Technology
chiba@is.titech.ac.jp

Yasushi Shinjo
University of Tsukuba
yas@is.tsukuba.ac.jp

Kazuhiko Kato
University of Tsukuba
kato@cs.tsukuba.ac.jp

Abstract

Virtual machine monitors (VMMs), including hypervisors, are a popular platform for implementing various security functionalities. However, traditional VMMs require numerous components for providing virtual hardware devices and for sharing and protecting system resources among virtual machines (VMs), enlarging the code size of and reducing the reliability of the VMMs.

This paper introduces a hypervisor architecture, called *parapass-through*, designed to minimize the code size of hypervisors by allowing most of the I/O access from the guest operating system (OS) to pass-through the hypervisor, while the minimum access necessary to implement security functionalities is completely mediated by the hypervisor. This architecture uses device drivers of the guest OS to handle devices, thereby reducing the size of components in the hypervisor to provide virtual devices. This architecture also allows to run only single VM on it, eliminating the components for sharing and protecting system resources among VMs.

We implemented a hypervisor called BitVisor and a *parapass-through driver* for enforcing storage encryption of ATA devices based on the parapass-through architecture. The experimental result reveals that the hypervisor and ATA driver require approximately 20 kilo lines of code (KLOC) and 1.4 KLOC respectively.

VEE'09, March 11–13, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-375-4/09/03. . . $5.00

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

General Terms Design, Security

Keywords Virtual Machine Monitors, Hypervisors, Trusted Computing Base, Parapass-Through, Shadow DMA Descriptor

1. Introduction

Security of desktop computers is becoming a serious problem. Compared to server computers, desktop computers are less managed and often vulnerable to various security attacks, while they tend to contain increasingly valuable information for personal and corporate users. Moreover, operating systems (OSs) for desktop computers, such as Windows and Linux, are becoming larger and more complex, resulting in inevitable security flaws in OS software. To address this problem, many researchers have proposed using virtual machine monitors (VMMs) for implementing security functionalities [4, 14, 2, 30, 13]. Since the execution environments of VMMs are isolated from guest OSs, VMMs can safely enforce security functionalities without depending on the untrusted OSs.

Using VMMs for enforcing security functionalities poses another problem: the security of VMMs themselves. If VMMs have a vulnerability, security functionalities in the VMMs might be disabled by attacks from the guest OSs. Reducing the code size of VMMs is an effective approach to improving reliability of VMMs [20]. Unfortunately, traditional VMMs including hypervisors are not necessarily small enough because they are designed for general purpose use and contain numerous components, such as device drivers and device models to provide virtual devices and resource managers to share and protect system resources among virtual machines (VMs). For example, the hypervisor of VMWare ESX Server has 200 KLOC (including device drivers) [28] and the

Xen hypervisor has 100 KLOC (excluding Domain 0) [20]. Especially, device drivers are less reliable than other components [6], degrading the reliability of VMMs.

Several recent researches have proposed small hypervisors for enforcing security [25, 5]. However, few of them address the security of I/O devices such as storage devices and network interfaces. Enforcing I/O device security in hypervisors is useful for preventing various security problems in desktop environments: enforcing encryption can prevent information leakage from the I/O devices regardless of the configuration of the guest OSs and hardware, and inspecting the contents of I/O devices can detect viruses and abnormal behavior even if the guest OSs are compromised. The purpose of our research is to design a hypervisor architecture dedicated to enforce I/O device security of desktop computers, thereby significantly reducing the code size of the hypervisors while they are practical enough to run commodity desktop OSs such as Windows.

In this paper, we introduce a hypervisor architecture, called *parapass-through*, designed to allow most of the I/O access from the guest OS to pass-through the hypervisor, while the minimum access necessary to implement security functionalities is completely mediated by the hypervisor. This architecture can eliminate most of the device drivers and device models from hypervisors by utilizing device drivers of the guest OS to handle real devices. Hypervisors only need a set of small drivers, called *parapass-through drivers*, to mediate access to I/O devices for enforcing security. The code size of parapass-through drivers is expected to be much smaller than that of traditional device drivers: they only need to monitor *control I/Os* to keep track of states of the devices, and intercept *data I/Os* to inspect or manipulate contents of I/O data. They do not need miscellaneous housekeeping functions such as initialization and error handling, which dominate most of the driver code [8]. Note that hypervisors does not trust guest device drivers: parapass-through drivers strictly enforce complete mediation of I/O devices and prevent the guest OS from bypassing them.

The parapass-through architecture also limits the number of VMs to one to allow direct access from the guest device drivers. We believe that running only a single VM, like other secure hypervisors [25], is a reasonable trade-off because the purpose of our hypervisor is to improve the security of desktop computers which usually need to run only a single OS in order to use office and Internet applications. Running a single VM also contributes to reducing the size of hypervisors because numerous components for sharing and protecting resources among VMs can be eliminated.

One of the most difficult issues with parapass-through drivers is the handling of Direct Memory Access (DMA). Since I/O data is directly transferred by DMA hardware, it is difficult for the hypervisor to intercept the I/O data without virtualizing the DMA hardware. To address this issue, this paper introduces a novel scheme using *shadow DMA descriptors*, allowing the hypervisor to intercept and manipulate data transferred via DMA, while most of the DMA access is still handled by the guest device drivers.

We have implemented a hypervisor, called *BitVisor*, based on the parapass-through architecture. BitVisor can run unmodified commodity OSs including WindowsXP/Vista and Linux, both in 32bit and 64bit mode, and runs on processors with Intel Virtualization Technology (Intel VT) [21] including multi-core processors. The experimental results reveals that the code sizes of the hypervisor and a parapass-through driver for encrypting ATA devices were approximately 20 KLOC and 1.4 KLOC respectively.

The contributions made by this paper include:

- Our proposal of a thin hypervisor architecture designed for enforcing I/O device security of desktop computers
- Our proposal of a novel scheme for intercepting DMA data by hypervisors without fully virtualizing devices, and
- Our implementation of a practical hypervisor based on the parapass-through architecture, which proved that the code size was much smaller than that in general-purpose hypervisors

This paper is organized as follows. Section 2 describes the threat model and assumptions underlying our hypervisor. Section 3 presents the architectures of traditional and parapass-through hypervisors. Section 4 describes the implementation of the parapass-through hypervisors and drivers in detail. Section 5 presents a case study of implementing a parapass-through driver for ATA devices. Section 6 presents the experimental results obtained by measuring the code size and performance of the hypervisors. Section 7 discusses related work and Section 8 summarizes our conclusions.

2. Threat Model and Assumptions

This section describes the threat model and assumptions we made in designing and implementing parapass-through hypervisors.

2.1 Threat Model

Potential threats for hypervisors implementing security functionalities include attempts to bypass the security functionalities by attacking the vulnerabilities of hypervisors. These attacks are divided into two types: software-based and hardware-based attacks.

We assumed that malicious attackers in software-based attacks could subvert and take over complete control of the guest OS. In this case, attackers could execute any kind of processor instructions in user and kernel mode. For example, an attacker could issue privileged instructions to access processor registers, memory access instructions to modify page tables, and various I/O instructions to control I/O devices. Examples of these attacks include issuing specially crafted I/Os to avoid security functionalities in the hypervisor, and accessing the memory regions or disk regions of the hypervisor by manipulating page tables or DMA controllers.

We assumed that malicious attackers in hardware-based attacks could only obtain physical access to computers when the computers were turned off. In other words, physical access to running hardware, such as probing buses and connecting malicious devices, was not assumed. In this case, we also assumed that computers that might have been physically accessed could be detected and users would not use compromised computers. For example, we did not assume attacks such as those modifying the disk image of the hypervisor by directly accessing the hard drives. This assumption corresponds to the situation where computers are lost or stolen.

We assumed that hardware and basic software such as firmware and BIOS could be trusted. Covert channels and other indirect communication channels that can be used to steal information from computers are beyond the scope of this paper.

2.2 Assumption

We assumed that the processors would have hardware support for virtualization such as Intel VT and AMD Virtualization (AMD-V) [7]. Old IA-32 processors without virtualization support were excluded since they are known to be inappropriate for running secure virtual machines [23]. Although we describe our implementation on Intel VT processors, most of the descriptions can be applied to other architecture processors. We also assumed that the platforms would have an input/output memory management unit (IOMMU) hardware. IOMMUs are used to prevent attacks by using DMA of I/O devices that are not monitored by the hypervisor, allowing secure (full) pass-through access to unmonitored devices, such as graphics and sound cards.

The hypervisors and the guest OS could run on multi-core processors. This means that a malicious guest OS might simultaneously access a device, although it will not occur in normal (not malicious) operating systems to avoid inconsistency.

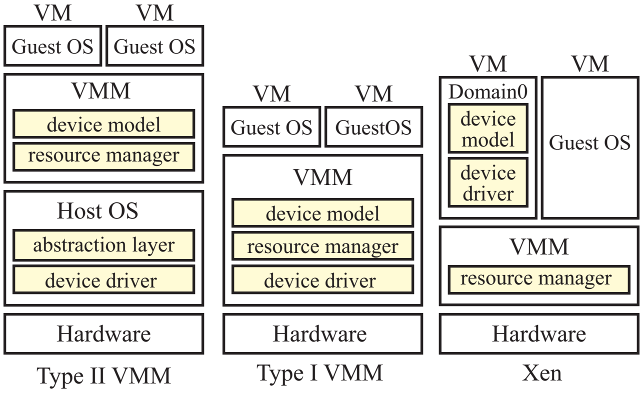


Figure 1. Traditional VMM architectures. Type II VMMs run on a host OS, while Type I VMMs run directly on the hardware. Both types of VMMs require device drivers and device models. Xen relocates device drivers and device models into a dedicated VM called domain0, which is still included in TCB. All of the VMMs require resource manager for sharing and protecting system resources among VMs.

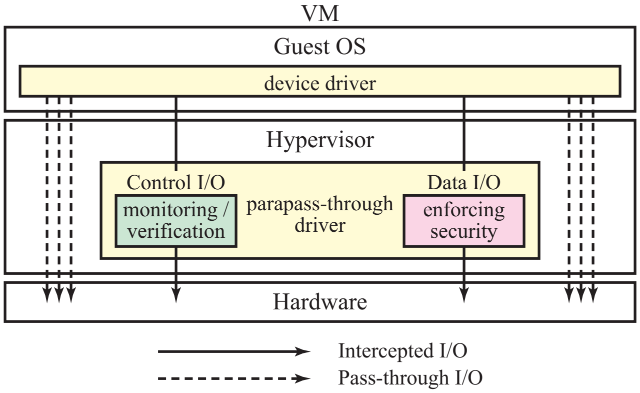


Figure 2. The parapass-through architecture. Most of the access to hardware from the guest device driver is pass-through. A part of I/Os is intercepted by the parapass-through driver in the hypervisor. The parapass-through driver intercepts control I/Os for monitoring and verification, and data I/Os for enforcing security such as encryption.

3. Design

This section describes the design of the parapass-through architecture. First, we explain traditional VMM architectures and analyze the code size of the VMMs and Trusted Computing Base (TCB). We then present the parapass-through architecture that is designed to minimize the code size of hypervisors.

3.1 Traditional VMM Architectures

The reliability of VMMs implementing security functionalities is an important issue. Since VMMs enforce security, they must be trusted and reliable, or there is no guarantee that the security functionalities will be effectively enforced. Reducing the code size of VMMs and the TCB it depends on is one of the most effective approaches to improving the reliability of VMMs. Although code size may not be the best metric to measure the reliability of software [20], it is widely used as one of the important metrics [9, 27, 8].

There are two types of VMM architectures: Types I and II [11]. Type II VMMs run on host OSs (see Figure 1). Since they depend on various functionalities of host OSs including file I/Os and network I/Os, the host OSs must be trusted and reliable. Therefore, the TCB of a Type II VMM includes both the VMM and (a part of) the host OS. For example, QEMU and Linux, a popular combination of a VMM and a host OS, respectively have 310 KLOC and 4,200 KLOC [29]. The code size of Type II VMMs and TCB tends to be much larger than that of other architectures because it includes various abstraction layers implemented in the OS kernel such as processes, files, sockets, and so on.

Type I VMMs, or hypervisors, run directly on the hardware and do not require host OSs, as shown in Figure 1. Instead of using host OSs, VMMs have device drivers and handle hardware devices by their own. This means that Type I VMMs must include numerous device drivers to support various devices such as hard drives, network interface cards, graphics, sounds, timers, and interrupts. They also require device models to provide virtualized devices to the guest OSs, and resource manager for sharing and protecting system resources among multiple VMs, as with Type II VMMs. As a result, Type I VMMs have become quite large; VMWare ESX Server's hypervisor called the VMkernel has 200 KLOC [28].

Xen [3] is a hypervisor that minimizes its size by moving device drivers and device models into a specific privileged VM, called Domain0 (see Figure 1). Xen also simplifies the device models through paravirtualization which defines simple interfaces between guest OSs and virtualized devices. Although the Xen hypervisor itself is relatively small (approximately 100 KLOC), it depends on various components including device drivers and device models running in Domain0 which should be included in TCB [20]. The software running in Domain 0 is a modified versions of Linux, significantly enlarging the size of the TCB of Xen as a whole.

In any of the traditional VMMs described above, the primary factors of increasing the size of VMMs are the existence of two major components: device managers (device drivers and device models) and resource managers. Our architecture is designed to shrink the size of device managers, which is after all included in TCB, and eliminate resource managers by giving up running multiple VMs, thereby significantly reducing the size of hypervisors.

3.2 Parapass-through Architecture

Let us introduce the *parapass-through* architecture. As shown in both sides of Figure 2, access from the guest OS to hardware devices basically passes-through the hypervisor. The guest OS handles the actual hardware devices of the computer as if it were directly running on the hardware. To allow direct access to the hardware, only a single VM can simultaneously run on parapass-through hypervisors. By abandoning support for running multiple VMs, parapass-through hypervisors eliminate numerous components needed for sharing and protecting resources among VMs. Moreover, this architecture allows the device drivers of the guest OS to handle real devices, allowing most of the device drivers and device models in the hypervisor to be eliminated.

If all the access is pass-through, the hypervisor is almost useless. Different from *fully* pass-through access, *para* pass-through hypervisors intercept a part of access to (1) protect hypervisors from the guest OS, and (2) enforce security functionalities. The access to be intercepted includes memory access and I/O access. Intercepting memory access is necessary to protect memory regions of the hypervisor and handle memory-mapped I/Os (MMIOs). Intercepting I/O access is necessary to protect the hypervisor and enforce security functionalities upon the I/Os for specific devices.

Intercepted I/O access is divided into two types: control I/Os and data I/Os (see Figure 2). Control I/Os are used to control data transfer between the guest OS and devices. They determine the location and size of data transferred to/from devices, the direction of transfers, and the start/stop of transfers. By intercepting control I/Os, the hypervisor can know the timing and details of data transfers. Some sensitive control I/Os are verified by the hypervisor to protect itself and blocked if they are unauthorized. For example, access to disk regions and memory regions used by the hypervisor is prevented. After verification, control I/Os are usually just recorded and passed to the device without modifications.

Data I/Os are used to transfer the data content. The hypervisor mediates data transferred between the guest OS and devices by intercepting data I/Os. By buffering data I/Os in the hypervisor, it can inspect and manipulate the content of data necessary to implement security functionalities, such as encryption or intrusion detection. Combined with the records of control I/Os, the hypervisor can accurately identify data to be intercepted.

3.3 Parapass-through Driver

Hypervisors in the parapass-through architecture need small drivers to handle intercepted I/Os. We have called these drivers *parapass-through drivers*. Parapass-through drivers know the specifications of target devices and correctly handle control I/Os and data I/Os. The drivers pass I/O data to the security functionalities after extracting and formatting the data in appropriate form, such as sectors in hard drives, and packets in network interface cards.

Parapass-through drivers correspond to device drivers and device models in traditional VMMs. It is important to keep device drivers and device models small because they are included in TCB. Unfortunately, conventional device drivers contains a great deal of code to handle miscellaneous housekeeping functions such as initialization, cleanup, management, and error handling. For example, 90% of the device driver of Intel's e1000 NIC is for such housekeeping functions and only 10% is essential for transferring data [8]. Parapass-through drivers, on the other hand, need to handle a small part of I/Os, which is essential for transferring data. We can expect that parapass-through drivers will usually become much smaller than conventional device drivers. In fact, our implementation of ATA parapass-through drivers was only 1.2 KLOC, while the libata driver of Linux 2.6 contains at least 9.0 KLOC. We can also expect that most of the functions of device models can be eliminated because they are handled by real devices.

Parapass-through drivers must be carefully designed to satisfy the principle of complete mediation [24]. This is a similar situation with reference monitors in sandbox systems that intercept system calls to protect against malicious software [10]. Detailed knowledge and thorough inspections of the specifications of the target devices are required to prevent omission. Fortunately, it is not so difficult because we only need to understand a part of the specifications, which is essential to transfer data. However, defining generic strategies to determine which I/O access should be monitored or intercepted is still our future work. This paper only shows a case study of ATA devices.

Parapass-through drivers are only required for I/O devices that need to be monitored. Therefore, the hypervisors usually require only a small set of parapass-through drivers. For example, if graphics and sound cards do not need to be monitored, parapass-through hypervisors can exclude drivers for those devices. I/O access to these devices from the guest OS completely passes through the hypervisor. This reduces the number of parapass-through drivers in the hypervisor and contribute to reduce the size and overhead of the hypervisor. We should note that DMA access by these devices are restricted by IOMMU hardware to prevent attacks that try to access the memory region of the hypervisor.

3.4 Limitations

Parapass-through hypervisors are basically passive: I/Os are only issued by the guest OS and hypervisors will not actively issue I/Os. This architecture limits the hypervisor functionalities that can be used to implement security functionalities. For example, hypervisors cannot use files or networks because they are under the control of the guest OS. However, the hypervisor can provide functionalities for allocating memory regions, intercepting I/Os, creating non-preemptive threads, and software-based timers [1]. These functionalities will be enough for implementing some classes of security functionalities: encryption is the best example and IDSs on storages or networks are also easy to implement. We plan to extend the architecture so that the security functionalities can actively access some I/O devices, thereby a broader class of security functionalities can be implemented in parapass-through hypervisors.

In the parapass-through architecture, security functionalities will run in hypervisors. Therefore, a vulnerability in the security functionalities becomes a vulnerability of the hypervisors. Unfortunately, moving the security functionalities into another protection domain will not necessarily solve the problem because they are still a part of TCB. Although improving the reliabilities of the security functionalities is still unsolved problem, we plan to implement lightweight protection domains inside hypervisors that may help mitigating the problem by isolating multiple security functionalities and hypervisors from each other.

4. Implementation

This section describes the implementation of parapass-through hypervisors, called BitVisor. We explain the handling of CPU, memory and I/O devices in turn.

4.1 CPU

We assumed that the processors would have hardware support for virtualization; the BitVisor hypervisor uses Intel VT processors. Hardware support is important for simplifying the implementation of hypervisors. We expected that processors would have two virtualization support functionalities:

- Saving/loading all necessary contexts of the processors and
- Transferring control to the hypervisor on sensitive events

Parapass-through hypervisors need two contexts: that of the guest OS and that of the hypervisor. Since only a single guest OS is supported, scheduling and context switches among guest OSs are unnecessary: context switches only occur between the guest OS and the hypervisor. On Intel VT processors, contexts are saved to and loaded from memory regions called VMCSs (VM control structures). These structures contain all processor states including registers that can not be accessed by processor instructions. Note that on multi-core processor systems, each processor needs its own contexts for the guest OS and hypervisor.

When a sensitive event occurs, such as execution of privileged instructions, the processor automatically switches the contexts from the guest OS to the hypervisor. The details on the events that cause control to be transferred to the hypervisor may depend on the processor architecture. BitVisor running on Intel VT processors currently captures the four types of events:

- the executions of some privileged instructions,
- exceptions and hardware interrupts,
- the executions of I/O instructions, and
- inter-processor interrupts (IPIs)

The privileged instructions to be captured include instructions to read and write control registers or model specific registers that may change the processor's sensitive states. Also, paging-related instructions such as invalidating a TLB entry must be captured because Intel VT processors do not yet have hardware support for virtualizing page tables like Nested Paging of AMD-V. How paging is handled is described in the next subsection.

Exceptions including page faults are also captured to handle paging. Hardware interrupts are used to implement timers that will be used in the future. BitVisor currently can not identity the devices that cause hardware interrupts due to the implementation problem in handling the Advanced Programmable Interrupt Controller (APIC), explained in the last subsection. Note that software interrupts do not need to be captured because they are handled by the processor and the guest OS.

I/O instructions need to be captured to enforce security on I/O devices. Intel VT processors support bitmaps that determine which I/O access is captured or not on a per port-address basis. BitVisor uses this bitmap to intercept I/O access to specific port addresses.

IPIs are used to start processors at boot time and signal events to other processors. The hypervisor needs to intercept start-up signals to initialize the virtualization functions of the processor before the guest OS starts, or the guest OS will start running directly on the processor without a hypervisor.

BitVisor emulates real mode by using virtual 8086 mode since Intel VT does not support virtualization of real mode. BitVisor also implements an instruction emulator to handle mode transitions between real mode and protect mode, and handle some I/O instructions and MMIOs to obtain details on the trapped instruction which the processor does not supply.

4.2 Memory

The guest OS of parapass-through hypervisors use the physical address space that is identical to the machine (real) physical address space. The hypervisor does not need to carry out address translations. This contributes to reducing the size of the hypervisor.

Translations from virtual addresses to physical addresses are carried out based on the page table of the guest OS. Unfortunately, the hypervisor cannot directly use the guest page table because the guest OS can access the memory regions of the hypervisor by setting a physical address of the memory regions to the page table. To prevent such attacks, the hypervisor must verify each page table entry before the entry is used by the processor.

BitVisor uses shadow paging similar to that is used in other VMMs [3] to verify page table entries. When a page fault occurs, the hypervisor sets the same entry to the shadow page table after the entry has been validated. The implementation of shadow paging is complicated and incurs significant overhead. Hardware support like nested paging of AMD-V [7] should significantly improve this situation; since physical to machine address translation is fixed in the parapass-through architecture, the hypervisor does not need to be involved in paging after initialization.

Hypervisors must hide their own memory regions from the guest OS so that the guest OS do not use them. BitVisor hooks the BIOS functions for obtaining the memory usage map (function e820h) to fake that the memory regions are reserved. BitVisor is currently located at a physical address under 4GB so that 32bit PCI devices can access the memory buffers in the hypervisor.

4.3 I/O devices

Parapass-through hypervisors need to intercept I/O accesses to enforce security on I/O devices. There are three types of I/Os: programmed I/Os (PIOs), memory mapped I/Os (MMIOs), and DMA. We will now explain methods of intercepting I/Os for these three types.

4.3.1 PIO

PIOs are carried out by dedicated I/O instructions, i.e., "IN" and "OUT" instructions on Intel processors. The I/O instructions specify an I/O port address to be accessed in a register. The hypervisors running on Intel VT processors can specify I/O port addresses to be intercepted using a bitmap, as previously described. Since each I/O device is allocated a range of I/O port addresses that will not overlap with other devices, hypervisors can intercept all the PIOs of the target device by specifying the addresses. I/O addresses may be changed by the PCI configuration mechanism. To capture the changes, hypervisors also intercept the PIOs of the PCI configuration mechanism and update the bitmap.

The units of PIO access are small, i.e., one, two, or four bytes. However, the hypervisor might need to handle I/O data in larger units. For example, storage encryption might need data for the entire sector (usually 512 bytes) as a unit of encryption, depending on the algorithm. In this case, the hypervisor temporarily buffers the PIO data until the unit data is read, then emulates I/Os so that the data is accessed in small units.

A guest OS running on a single processor is blocked on issuing I/O instructions. Therefore, a device is serially accessed from a guest OS. However, a guest OS running on multicore processors might try to simultaneously access a single device from multiple processors. This might cause time-of-check-to-time-of-use (TOCTTOU) problems: a guest OS running on a processor might change the status of a device that has just been verified by the hypervisor running on another processor. A normal guest OS will not attempt such access because it might cause unexpected behavior. However, a malicious OS may intentionally attempt. To avoid this problem, the hypervisor must hold a lock on the devices.

4.3.2 MMIO

MMIOs represent a method of accessing I/O devices with memory-access instructions. The registers of devices are mapped into a memory region in the physical address space and the registers can be read or written through it. OSs map the region into the kernel address spaces by using page tables.

MMIOs can be intercepted by using shadow paging. By setting shadow page tables so that the MMIO memory regions are inaccessible, control is transferred to the hypervisor when the guest OS tries to access the MMIO region. The difference from PIOs is that the unit of interception is a page: it is not possible to intercept only a part of a page. Although all necessary I/Os can be intercepted, this might cause performance problems by intercepting numerous unnecessary I/Os. Unfortunately, this is an unavoidable nature of intercepting MMIOs by hypervisors.

There is a timing problem with flushing TLB on multiprocessor systems. When the guest OS changes the physical address of an MMIO region, the hypervisor must update the shadow page table so that the new MMIO region can be intercepted. After the shadow page table has been updated, TLBs must be flushed so that old entries are no longer used. Unfortunately, it is not easy for parapass-through hypervisors to flush TLBs on all processors. If a guest OS running on a processor does not transfer control to the hypervisor, the hypervisor cannot flush TLBs on that processor. The hypervisor cannot use IPIs since IPIs are defined and controlled by the guest OS.

BitVisor handles this problem by scanning the shadow page table to ensure that the new MMIO region is not on the TLBs on any processor. Usually, a new region will not be on TLBs because this means the guest OS has accessed unused regions. If the region is not on the TLBs, the hypervisor does not need to flush them. If the region is on the TLBs, the hypervisor waits to obtain control of all processors.

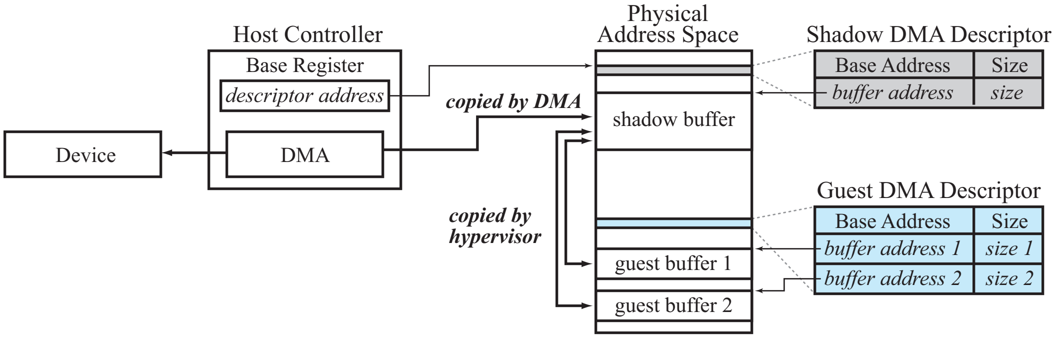


Figure 3. The scheme using shadow DMA descriptors. When data are read from the device, the data are first copied to the shadow buffer via DMA. The hypervisor then copies the data to the guest buffers based on the guest DMA descriptor. When data are written to the device, the data are first copied from the guest buffers to the shadow buffer by the hypervisor. The data are then written to the device via DMA. The hypervisor can manipulate data on the shadow buffer. The host controller is mostly handled by the device driver of the guest OS.

4.3.3 DMA

DMA is a mechanism to transfer data between devices and memory without processor intervention. A DMA host controller is connected to the device and the memory bus, and automatically transfers data between them. Modern DMA host controllers use *DMA descriptors*, a memory region that describes transfer information, such as the buffer address and the size of the data. A DMA descriptor contains a series of entries and the host controller carries out data transfers one by one based on the entries.

Hypervisors can access data transferred via DMA by directly reading or writing the guest memory regions. However, it is not a good scheme for enforcing security. Since DMA transfers are carried out in parallel with the execution of guest OSs, they can access a part of data before the transfer has been completed. Moreover, guest OSs can modify the entries of DMA descriptors while DMA transfers are carried out. Therefore, the guest OS might be able to bypass verifications or manipulations of DMA data by hypervisors.

This paper introduces a novel scheme using *shadow DMA descriptors* (see Figure 3) for safely intercepting the content of data transferred by DMA. A shadow DMA descriptor is a shadow of the DMA descriptor of the guest OS (*guest DMA descriptor*). Similar to shadow page tables, the hypervisor sets the shadow DMA descriptors to the host controller instead of the guest DMA descriptor. The shadow DMA descriptor specify a memory region in the hypervisor as a buffer, called the *shadow buffer*. The DMA host controller transfers data between the shadow buffer and the device, based on the shadow DMA descriptors. The hypervisor emulates the host controller by copying data between the shadow buffer and the *guest buffer* that is specified in the guest DMA descriptor.

The hypervisor can inspect and modify the data transferred by DMA because the data is buffered in the shadow buffer. For example, the hypervisor can encrypt or decrypt the data, or perform intrusion detection on the data. Since the shadow buffer and the shadow DMA descriptor is located in the hypervisor's memory region, the guest OS cannot access them. By shadowing DMA descriptors and buffers, the hypervisor can prevent the guest OS from reading data before or writing data after the inspection by the hypervisor.

There is a difference between the handling of DMA read and write operations. When data are read from the device, they are first transferred from the device to the shadow buffer by DMA; then, the data are copied to the guest buffer by the hypervisor. The hypervisor needs to capture the events to start DMA transfer. When data are written to the device, on the other hand, the data are first copied from the guest buffer to the shadow buffer by the hypervisor, and then the data are transferred to the device. The hypervisor needs to capture the events of notifying the end of DMA transfer.

Starting DMA usually requires PIO or MMIO access to a device register. Therefore, the hypervisor can easily capture the event. The end of DMA transfer is usually notified by a hardware interrupt. Unfortunately, the current BitVisor cannot identity the device that issues hardware interrupts. Instead, BitVisor captures I/O access to status registers, because device drivers usually read status registers to check whether DMA transfer has finished successfully or not, and write registers to acknowledge interrupts.

The hypervisor must prevent attacks that try to modify its memory regions by using DMA. The hypervisor verifies the address of guest buffers specified in the guest DMA descriptors so that the address does not point to the hypervisor regions. This technique is similar to that used in a software-based IOMMU [26]. To prevent attacks using DMA that do not need to be intercepted, we can use IOMMU hardware.

4.4 Miscellaneous Issues

Identifying devices that cause hardware interrupts is difficult for parapass-through hypervisors. Although the hypervisor can capture hardware interrupt events, it must know the mapping from the vector number of the interrupt to the device that issues the interrupt. This requires the APIC to be monitored. Unfortunately, since APIC is very frequently accessed via MMIO, monitoring APIC incurs a significant overhead. BitVisor ceases identifying hardware interrupts and captures access to status registers instead.

Modern OSs use an Advanced Configuration and Power Interface (ACPI) to manage power. Guest OSs running on parapass-through hypervisors handle ACPI to suspend and resume computers. Therefore, the entry point where control is transferred when the computer resumes is set by guest OSs. Unfortunately, hardware-virtualization functions are turned off when processors are suspended. As a result, the guest OS obtains control bypassing the hypervisor when the computer resumes. To support suspend and resume functions, the hypervisor needs to handle ACPI. BitVisor does not implemented this yet.

5. Case Study: ATA Host Controller

This section presents a case study where storage encryption was enforced in BitVisor. We implemented a parapass-through driver for standard ATA host controllers. We explain the handling of ATA host controllers and DMA descriptors in the following.

Command Block Registers

Offset	Bit Count	Register Name (read)	Register Name (write)
0	16 bit	Data	Data
1	8 bit	Error	Features
2	8 bit	Sector Count	Sector Count
3	8 bit	LBA Low	LBA Low
4	8 bit	LBA Middle	LBA Middle
5	8 bit	LBA High	LBA High
6	8 bit	Device	Device
7	8 bit	Status	Command

Control Block Registers

Offset	Bit Count	Register Name (read)	Register Name (write)
0	8 bit	Alternate Status	Device Control

Bus Master (DMA Host Controller) Registers

Offset	Bit Count	Register Name
0	8 bit	Command (DMA)
1	8 bit	Reserved
2	8 bit	Status (DMA)
3	8 bit	Reserved
4-8	8 bit * 4	Descriptor Pointer (DMA)

Figure 4. The registers in ATA host controllers

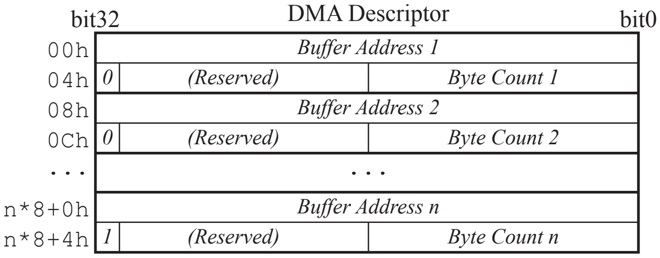
5.1 Host Controller

ATA host controllers use eight command-block registers and one control-block register. Command-block registers include registers that specify logical block addresses (LBAs), sector counts, and commands such as READ and WRITE. When PIO transfer is used, a 16 bit data register is utilized. Figure 4 shows the layout of the registers in the I/O address space.

As previously described, the hypervisor can intercept I/O access to these registers. Therefore, it is easy to obtain the information necessary to enforce encryption. For example, the hypervisor can obtain the LBA and sector count by intercepting writes to these registers. The hypervisor can also obtain the direction of transfer by intercepting values written to the command register (e.g., 0x20 is ”READ SECTOR” and 0x30 is ”WRITE SECTOR”).

However, intercepting I/O access to ATA registers may cause a TOCTTOU problem. To prevent unauthorized access to sectors from the guest OS, the hypervisor must check the LBA value the guest OS requests. Unfortunately, ATA registers are not necessarily always writable. For example, the ATA specification states that LBA registers are not writable when the device is busy or in sleep mode. If the hypervisor does not know what state the device is in, the value written by the guest OS may not be actually written to the registers and the old value may be retained, while the hypervisor believes that a new value has been written to the device. To avoid this problem, BitVisor simply reads the register value just before the guest issues a ATA command, such as “WRITE SECTOR”, which lets the device hardware to initiate data transfer using the register value. Reading the registers instead of intercepting writes to the registers allows keeping consistency between the register values the hypervisor knows and that of the real hardware.

This strategy works fine if the registers are readable. Unfortunately, some registers are write-only. For example, the command register and the status register are assigned to the same I/O address and the command register is write-only (see Figure 4). In some commands, the features register, a write-only register, is used to specify part of the LBA. In this case, BitVisor rewrites the value that is saved in the hypervisor by intercepting writes to the register just before the command is written.



	bit32	DMA Descriptor	bit0
00h	*Buffer Address 1*		
04h	*0*	*(Reserved)*	*Byte Count 1*
08h	*Buffer Address 2*		
0Ch	*0*	*(Reserved)*	*Byte Count 2*
. . .	. . .		
n*8+0h	*Buffer Address n*		
n*8+4h	*1*	*(Reserved)*	*Byte Count n*

Figure 5. The structure of the DMA descriptor in ATA devices

5.2 DMA descriptor

The structure of the DMA descriptor in ATA devices is simple, as seen in Figure 5. It is an array of structures, each of which contains a physical address of the buffer, the transfer size, and the end of the table bit. The address of the DMA descriptor is set to a host controller register. When the start bit in the DMA command register is set, the controller starts transfers by reading each entry of the DMA descriptor until it encounters the end bit set. Data can be scattered to or gathered from buffers in physical memory.

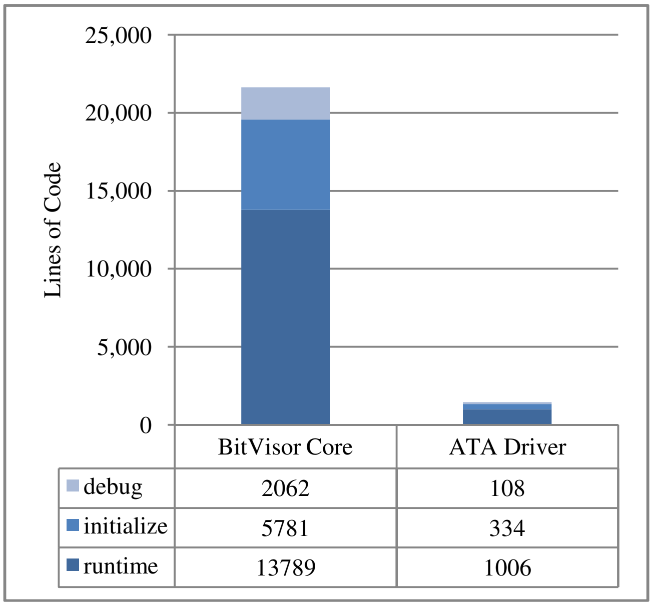
In the parapass-through architecture, the hypervisor creates a shadow DMA descriptor and set to the descriptor pointer register of the host controller. The hypervisor also remember the address of guest DMA descriptor by intercepting access to the descriptor pointer register. The shadow DMA descriptor points to a single pre-allocated shadow buffer in the hypervisor. Before transfers, the hypervisor obtains the total size of data transfers by scanning the guest DMA descriptor and set the size to the first entry of the shadow DMA descriptor. After transfers, the hypervisor emulates the host controller by copying data between the shadow buffer and guest buffers. The size of the shadow buffer is currently 512 KB. If a single transfer exceeds that size, the transfer must be splitted into multiple transfers, which is not implemented yet. Fortunately, 512 KB is sufficient for the normal workloads of Windows and Linux.

6. Experiments

This section presents the experimental results obtained by measuring the code size and performance of BitVisor. BitVisor is a practical implementation of the papapass-through architecture: it supports Intel VT processors including multi-core processors running in 32 bit mode or 64 bit mode. BitVisor includes a parapass-through driver for ATA devices that has a function to encrypt data with XTS-AES [12]. BitVisor can run WindowsXP/Vista, Linux, and FreeBSD as a guest OS. However, the implementation for using IOMMU is not stable yet and the IOMMU hardware is not used in the following experiments.

6.1 Code Size

We used SLOCCount [29] to count the effective code size of the hypervisor. Figure 6 shows the results. Although the core of the BitVisor has 21,582 lines of code in total, the runtime code excluding debug and initialization code has only 13,789 lines of code. The runtime includes 1,130 lines of code for memory management based on shadow paging, and 2,239 lines of code for the instruction emulator. The experimental results revealed that the code size of parapass-through hypervisors is significantly smaller than that of other hypervisors, such as the Xen hypervisor having 100 KLOC [20]. The small code size contributes to the reliability of the hypervisor. Note that the code for memory management is expected to be much smaller by using hardware assisted paging instead of shadow paging.



	BitVisor Core	ATA Driver
▨ debug	2062	108
▨ initialize	5781	334
■ runtime	13789	1006

Figure 6. Source Lines of Code of BitVisor

Host	null	fork	exec	prot	page	ctx
Linux	0.21	93.4	96.6	0.36	0.98	1.10
BitVisor	0.21	2859	3049	2.73	34.6	42.2

Table 1. Execution times of lmbench proc. & mem. (μs)

The ATA parapass-through driver is implemented by using 1,448 lines of code (not including XTS-AES code). The runtime only has 1,006 lines of code. For comparison, we measured the code size of the libata, an ATA driver included in Linux 2.6.26, and found that it has 9.9 KLOC for files ”libata*”. Surprisingly, this result coincides with the study of the Intel’s Ethernet driver: only 10% of the code directly relates to transfering data [8]. In addition, conventional VMMs need an implementation of the device model for ATA devices. The result proved that the size of parapass-through drivers becomes significantly smaller than that of conventional device drivers and device models.

6.2 Performance

We will next discuss the overhead of the parapass-through hypervisor. We carried out experiments on a machine with an Intel Core 2 Duo E6850 (3.0 GHz), having 2 GB memory, and a 74 GB 10,000 rpm hard disk drive (Western Digital Raptor WD740GD). We used the 32 bit version of Fedora 8 (the kernel was Linux 2.6.25.9-40.fc8) and the 32 bit version of Windows XP as guest OSs. BitVisor was compiled with the 64 bit mode.

lmbench overhead We used lmbench benchmark suite [17] in the first experiment to measure the overhead of the operations in the guest OS. BitVisor is expected to incur overheads on memory management operations because it uses shadow paging, causing control transfer to the hypervisor on every page fault.

Table 1 lists the results obtained from the lmbench experiments. The `null` indicates the overhead of a null system call. The hypervisor does not need to intercept events for entering and leaving the kernel because this is automatically handled by the processor hardware. Therefore, there is no overhead for `null`. The `fork` and `exec` indicate the overhead for each system call. Creating processes and executing a program require operations on page tables, which incur an overhead. The `prot` indicates the overhead of a protection fault. A protection fault incurs a single round trip to the hypervisor to verify that it is a protection fault. Then, the hypervisor returns control to the kernel and the handling of the protection fault is continued. The `page` indicates the overhead of a page fault. A page fault, in contrast to a protection fault, incurs two round trips to the hypervisor to first verify that it is a page fault and then the shadow page table is updated after the guest kernel has inserted the entry into the guest page table. Therefore, the overhead of `page` is higher than that of `prot`. The `ctx` indicates the overhead of a context switch between processes. In the case of context switches, the hypervisor needs to intercept the context switches, switch shadow page tables, and handle the following page faults that follow. As a result, the overhead of a context switch becomes close to that of a page fault.

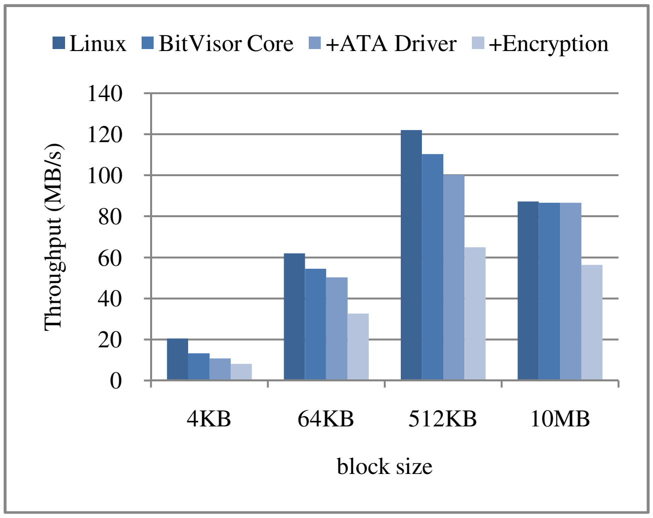


Figure 7. Storage Overhead

Storage overhead We measured the overhead for intercepting and encrypting storage data. We used lmdd in the lmbench suite to measure the throughput for reading data from the hard disk drive. We used a device file of Linux that was opened by lmdd with the flag `O_DIRECT`, meaning that buffer cache of the operating system kernel was bypassed. The read data was just discarded.

Figure 7 shows the results. The left-most bar indicates the throughput for native Linux. The `BitVisor Core` means that only the hypervisor core is running and all I/O access to ATA devices is fully pass-through. The `+ATA driver` means that the hypervisor core and the ATA parapass-through driver are enabled but the driver does not execute encryption or decryption, i.e., data is intercepted but just passed through. The `+Encryption` means that the hypervisor core, the ATA parapass-through driver, and encryption/decryption by AES-XTS are enabled. We used Dr. Brian Gladman’s AES engine written in 64 bit assembler.

Each of `BitVisor Core`, `+ATA Driver`, and `+Encryption`, up to 512KB, incurs overhead. Although the buffer cache of the kernel is bypassed, the hard disk drive itself has a cache on the controller. Therefore, actual media access is not carried out up to 512 KB. On the other hand, a 10 MB read does not demonstrate an overhead for the `BitVisor core` or `+ATA Driver` because the overhead is hidden by slow media access. Nevertheless, `+Encryption` still incurs an overhead because the software execute encryption, which requires a great deal of computation: encrypting one sector

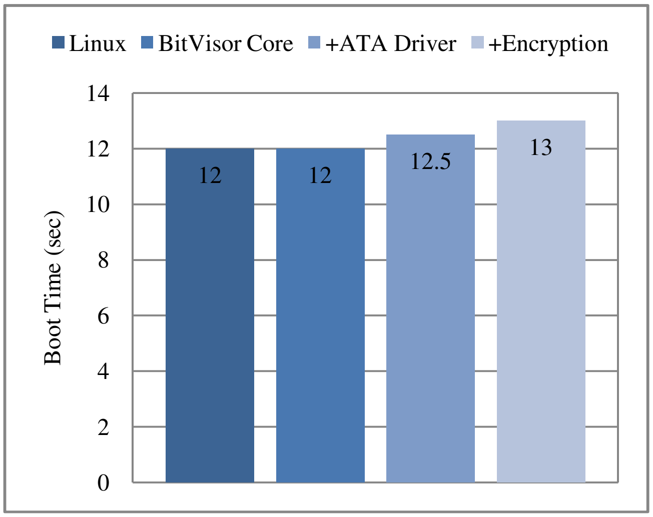


Figure 8. Windows XP Boot Time

(512 bytes) required 6995 cycles and decrypting required 7038 cycles (about 2.3 μs on our machine). When throughput is 80 MB/s, 512 bytes can be transferred in 6.4μs. Therefore, 2.3μs represents a 36% overhead, which corresponds to the result.

Windows Boot Time We measured the time to boot Windows XP to which only standard hardware drivers are installed.

Figure 8 shows the results. The hypervisor does not significantly affect the Windows XP boot time: the difference is less than 1 second even if storage encryption is enabled. This is because Windows XP will wait for the initialization of I/O devices at boot time, and the overhead for memory management and storage I/Os is hidden behind the wait time. The results show that although the BitVisor hypervisor incurs overhead on memory management operations and storage access, the impact on users of desktop computers is not expected to be so high and the hypervisor is practical enough to be used in normal desktop environments.

7. Related Work

Several recent researches have proposed tiny hypervisors to implement security functionalities. The design of these hypervisors are dedicated to security instead of running multiple virtual machines, reducing the code size of the hypervisors by eliminating functionalities for sharing and protecting system resources among VMs. However, most of these researches use memory management for enforcing security and do not address security in I/O devices.

SecVisor [25] uses a tiny hypervisor to preserve the integrity of kernel code. To prevent unauthorized code from being executed in kernel mode, SecVisor ensures that only approved code is executable in kernel mode by verifying modifications to the page tables. SecVisor also uses IOMMU to prevent approved code from being modified by the DMA write. The concept and techniques used in SecVisor to implement a tiny hypervisor are similar to those of BitVisor. However, SecVisor does not intercept the data of device I/Os and most of these are completely pass-through.

Overshadow [5] uses a hypervisor to prevent information leakage from trusted applications to untrusted operating system kernels. Overshadow presents encrypted view of pages to the kernel while presenting unencrypted view of the page to trusted applications. Overshadow intercepts page access and transparently encrypts/decrypts the page data. Although Overshadow uses VMware, the authors say that a simpler hypervisor can be used. However, Overshadow needs to recognize the semantics of guest operating systems to handle multiview pages, while BitVisor is transparent to the guest OS.

Some researches have used small VMMs to improve management of machines. LVMM [22] uses a VMM that runs two VMs on desktop machines: a user partition VM for running a standard desktop operating system and a services partition VM for a small management operating system. LVMM allows the the user partition VM to directly handle most devices except for an NIC device. The services partition VM virtualizes the NIC device to allow concurrent access to this NIC device from the user and services partition VM. LVMM uses device drivers and device models for NIC devices in the services partition VM, while BitVisor uses a parapass-through driver, which is much smaller than typical device drivers.

Microvisor [16] supports devirtualizable virtual machines. It runs two operating systems while maintenance to reduce downtime. When the maintenance has finished, applications running on the old guest operating system are migrated to the new operating system and it is devirtualized to run directly on the hardware. Microvisor splits hardware resources like memory and devices into two and they are dedicated to each VM. The VMs have completely direct access to the dedicated device. BitVisor allows the VM direct but verified access to I/O devices.

Type-II VMMs have been used in several researches and products to implement security functionalities [18]. Using Type-II VMMs allows security functionalities to be easily developed by utilizing the rich functionalities of host OSs. However, the code size of its TCB is much bigger than that of Type-I VMMs.

Several researchers have tried to improve the performance of device I/Os by allowing direct access to the device from a specific VM [15]. This approach can eliminate device models and device drivers for specific devices. However, in this case, the VMMs can not intercepts I/O accesses to the devices. Therefore, it is difficult to enforce security on I/O data.

Shafer et al. [26] proposed a software-based IOMMU that intercepts and verifies the DMA descriptor to allow direct access to the NIC device from a guest OS. This technique is similar to the shadow DMA descriptor used in BitVisor. However, the software-based IOMMU only performs access control for protection, while BitVisor needs shadowing of the DMA descriptor to capture the content of data transferred by DMA.

Microdrivers [8] split a device driver into two parts: only a performance critical part runs in kernel mode and the other infrequently executed part runs in user mode. In additoin, Microdrivers can automatically split existing code of device drivers. Microdrivers and BitVisor are similar in that they run only a performance or security critical part runs in a trusted domain. However, the functions of parapass-through drivers include not only a part of device drivers but also a part of device models, and the actual implementation of the drivers is significantly different from existing code. Therefore, we can not simply extract code for parapass-through drivers from existing device drivers or device models.

8. Conclusions

This paper described the design and implementation of a hypervisor for enforcing security in I/O devices. A VMM architecture called parapass-through is introduced to improve reliability. Parapass-through hypervisors intercept only a small set of hardware access that is necessary for enforcing security, while other access is mostly pass-through. This paper also proposed a novel scheme using shadow DMA descriptors for capturing I/O data transferred by DMA. The parapass-through architecture, combined with the shadow DMA descriptor scheme, significantly reduced the code size of hypervisors. The experimental results revealed that the code size of a parapass-through hypervisor was approximately 20 KLOC and 1.4 KLOC for the ATA parapass-through driver, which are much smaller than those for traditional VMMs.

In the future, we plan to apply the parapass-through architecture to other I/O devices like USB devices and Ethernet devices. These devices have different structures and interfaces of DMA descriptors, complicating the shadowing of the DMA descriptors. However, we have already implemented prototype parapass-through drivers for these devices and we believe that the architecture can be applied to various devices. We also expect that parapass-through hypervisors can be used as a basis for implementing various security functionalities [19]. We plan to extend the architecture so that richer class of security functionalities can be easily deployed.

Acknowledgments

This work was supported by Special Coordination Funds for Promoting Science and Technology from MEXT, Japan.

References

[1] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 232–246, December 1999.

[2] Kurniadi Asrigo, Lionel Litty, and David Lie. Using vmm-based sensors to monitor honeypots. In *Proc. of the 2nd International Conference on Virtual Execution Environments*, pages 13–23, June 2006.

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

[4] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems* (*HotOS-VIII*), pages 133–138, May 2001.

[5] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, March 2008.

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, and Dawson Engler. An empirical study of operating systems errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001.

[7] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming rev 3.14, September 2007.

[8] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, March 2008.

[9] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[10] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proc. of the 6th USENIX Security Symposium*, July 1996.

[11] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[12] IEEE. IEEE standard for cryptographic protection of data on block-oriented storage devices, April 2008. IEEE Std 1619-2007.

[13] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proc. of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 91–100, March 2008.

[14] Kenichi Kourai and Shigeru Chiba. HyperSpector: Virtual distributed monitoring environments for secure intrusion detection. In *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 197–207, June 2005.

[15] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proc. of the 2006 USENIX Annual Technical Conference*, pages 29–42, May/June 2006.

[16] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–233, October 2004.

[17] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. of the 1996 USENIX Annual Technical Conference*, January 1996.

[18] R. Meushaw and D. Simard. Nettop: Commercial technology in high assurance applications, 2000.

[19] Junichi Murakami. A hypervisor IPS based on hardware assisted virtualization technology. In *Black Hat USA 2008*, August 2008.

[20] Derek G. Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *Proc. of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 151–160, March 2008.

[21] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(03):167–177, August 2006.

[22] Mahendra Ramachandran, Ned Smith, Matthew Wood, Sharad Garg, Jim Stanley, Eswar Eduri, Rinat Rappoport, Arie Chobotaro, Carl Klotz, and Lori Janz. New client virtualization usage models using intel virtualization technology. *Intel Technology Journal*, 10(03):205–216, August 2006.

[23] John Scott Robin. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proc. of the 9th USENIX Security Symposium*, August 2000.

[24] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[25] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*, pages 335–350, October 2007.

[26] Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, Willy Zwaenepoel, and Paul Willmann. Concurrent direct network access for virtual machine monitors. In *Proc. of the IEEE 13th International Symposium on High Performance Computer Architecture*, pages 306–317, February 2007.

[27] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 161–174, April 2006.

[28] VMWare. Vmware esx server virtual infrastructure node evaluator's guide, November 2005. `http://www.vmware.com/pdf/esx_vin_eval.pdf`.

[29] David A. Wheeler. Counting source lines of code (sloc). `http://www.dwheeler.com/sloc/`.

[30] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 71–80, March 2008.