Guardian: Hypervisor as Security Foothold for Personal Computers

Yueqiang Cheng and Xuhua Ding

School of Information Systems
Singapore Management University
{yqcheng.2008,xhding}@smu.edu.sg

Abstract. Personal computers lack of a security foothold to allow the end-users to protect their systems or to mitigate the damage. Existing candidates either rely on a large Trusted Computing Base (TCB) or are too costly to widely deploy for commodity use. To fill this gap, we propose a hypervisor-based security foothold, named as *Guardian*, for commodity personal computers. We innovate a bootup and shutdown mechanism to achieve both *integrity* and *availability* of Guardian. We also propose two security utilities based on Guardian. One is a device monitor which detects malicious manipulation on camera and network adaptors. The other is hyper-firewall whereby Guardian expects incoming and outgoing network packets based on policies specified by the user. We have implemented Guardian ($\approx$ 25K SLOC) and the two utilities ($\approx$ 2.1K SLOC) on a PC with an Intel processor. Our experiments show that Guardian is practical and incurs insignificant overhead to the system.

1 Introduction

The operating system is the cornerstone of all security applications such as anti-virus and firewall. Once the OS is compromised, the adversary has the ability to disable all security services and access all sensitive data in the system. Even if a security-conscious end-user is aware of the attack, she still can not get a reliable security foothold to mitigate the damage. It is challenging to seek a feasible and secure solution. Rewriting commodity OS, e.g., splitting the OS into low- and high-assurance portions, is too costly to be practical. Adopting new security-capable devices (e.g., secure co-processors) usually requires substantial modifications to hardware, OS and/or applications, which makes these solutions difficult to widely deploy in the near future.

A usable security foothold should meet the following requirements. Firstly, it should be secure against attacks from rootkits which can subvert the operating system. Secondly, it should allow the human user to use it, e.g., to issue a command. Last but not the least, it should be always available throughout the life cycle even when the OS is corrupted. By virtue of the virtualization, a hypervisor is widely deemed as a software which can resists attacks from an untrusted guest OS. However, almost no hypervisor can simultaneously satisfy all the above requirements, especially for the availability requirement.

M. Huth et al. (Eds.): TRUST 2013, LNCS 7904, pp. 19–36, 2013.

In this paper, we harness the fast-growing hardware-assisted virtualization techniques to build a tiny but reliable hypervisor as the security foothold for personal computers. The hypervisor we propose is named as *Guardian*. Guardian has two prominent new features which are the enabling techniques for the hypervisor to become a security foothold. The first is a new secure bootup and shutdown mechanism, which enhances the existing hardware-based security boot up by offering integrity and availability protection of the TCB image and critical information. The other feature is a secure user-hypervisor interface which allows the end-user to issue commands to and receive responses from Guardian at runtime. The interface is secure in the sense that the channel between the human end-user and the hypervisor is authentic and the exchanged information is not exposed to the guest. We also propose two practical security utilities based on Guardian. The first is a device monitor utility, whereby the user can instruct Guardian to monitor the state of peripheral devices, e.g., a camera. The second is a hyper-firewall whereby Guardian inspects inbound/outbound network traffic and drops illegal packets. We have implemented Guardian on a desktop with a Linux guest. Guardian consists of around $25K$ SLOC, and the utilities consist of around $2.1K$ SLOC. Our experiments show that Guardian inflicts an insignificant workload to the whole system.

The growing hardware support for virtualization will continue to empower the hypervisor with more effective and stronger security control over commodity platforms with smaller code size and better performance. We envisage that using a hypervisor as a generic security foothold is a promising direction to greatly boost up the security for commodity platforms. Our work presented in this paper is an important step towards this ultimate goal. We summarize our contributions as follows:

1. We design and implement Guardian which is the first system to provide both *integrity* and *availability* guarantees. Note that all existing hypervisors do *not* achieve the availability guarantee.
2. We design and build a device monitor and a hyper-firewall as two security utilities on top of Guardian.

In the next section, we present our research objectives and threat model. Then we present the design of Guardian and the security utilities in Section 3 and Section 4. In Section 5, we describe the implementation and the evaluation. Finally, we discuss the related work in Section 6 and conclude the paper in Section 7.

2 Problem Definition

We aim to provide a tiny and reliable hypervisor as a security foothold for personal computers. Namely, we undertake to furnish the end-user with a reliable security basis when the conventional one (typically the operating system) fails. Though the security foothold, the human user can configure security policies and manage resources in the platform. It not only boosts up the system security, but also facilitates the end-user to determine the trustworthiness of her system. Note that we do *not* attempt to detect and remove malicious software from the platform, nor is to protect the operating system or a user application.

2.1 Threat Model

Since our goal is to assist the end-user, we assume that they are security- conscious users, who are happy and intended to use our system to protect their systems. We do not consider any human adversary who may have physical access to the system. For instance, the adversary can issue malicious DMA accesses by inserting extra physical devices (e.g., a firewire device). A malicious human user can always remove the hypervisor from the platform.

The adversary in our threat model is malware residing in the operating system which can subvert the operating system and launch arbitrary attacks. However, we assume that they can not compromise the hypervisor. Note that the hypervisor makes use of hardware-assisted virtualization techniques to defend against malicious software accesses and illicit DMA accesses. This assumption can be more reasonably held if the hypervisor has a tiny code size and simple logic so that only a small attack interface is exposed to the adversary. Existing techniques [4, 24, 39, 40] can also be applied to enhance hypervisor security.

We assume that the adversary can not compromise the hardware devices whose behavior always exactly follow their specifications. We also assume the system firmware is trusted. In fact, the modern BIOS has a built-in hardware lock mechanism [17, 35] to set itself as read-only so that the OS cannot tamper with it. Furthermore, the modern BIOS only accepts signed updates [36, 38]. Due to the complexity of the x86 platform (e.g., optional ROM), this assumption may not always true. Nonetheless, it is still possible to validate the system firmware by the proposed attestation approach [20] or by a trusted system integrator.

3 Design of Guardian

In this section, we introduce the techniques for establishing Guardian as a security foothold, and describe the functionalities of the two secure user interfaces.

3.1 Establishing Guardian as a Security Foothold

To establish Guardian as a security foothold, it is necessary but not sufficient to ensure a secure boot. The secure boot alone can only validate the integrity of the system's TCB image during booting up, while a reliable security foothold needs both integrity and availability guarantee, so that the system still boots up into a trusted state even if the TCB image on the hard drive are modified by attackers. We do not elaborate the details of secure boot (e.g., TPM-based secure boot [37]) to avoid verbosity as it has been widely used in the literature. Our focus is to explain how to ensure that the intact TCB image is always available for the boot up. The TCB of our system consists of the BIOS, the bootloader-core and the Guardian image. Recall that the BIOS is protected by the hardware and is trusted in our threat model. Therefore, we intend to protect the bootloader core and the Guardian image against runtime attacks.

A straightforward approach is for Guardian to intercept and validate every disk I/O, such that any access to the security critical image residing on the disk is blocked.

Obviously, this solution is costly due to the high overhead and complexity of a disk I/O interception multiplied by the huge number of disk operations.

We devise a novel scheme without interposing on disk operations. The basic idea (visualized in Figure 1) is that once Guardian is launched, it immediately relocates its image and the bootloader core from the disk into a protected memory region *prior to* launching the guest. Then, Guardian intercepts all power off events, and writes the protected image back to the disk before cleaning up the memory. In the following, we describe the details of secure boot up and secure shutdown, which in tandem with runtime protection bolster the availability of Guardian throughout its whole life cycle.

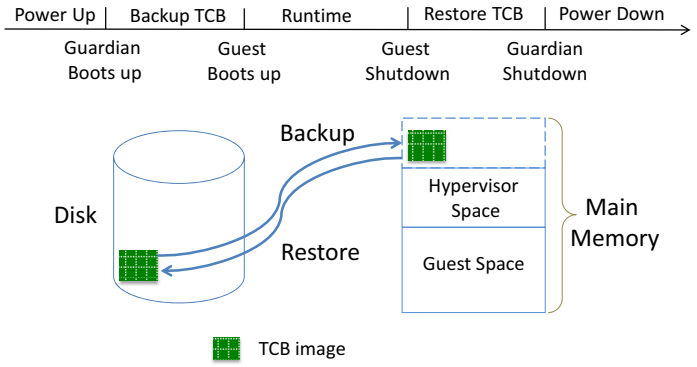

Fig. 1. Protection of the TCB (from power up to power down). The TCB consists of the Guardian image and the bootloader core. The protected memory for the TCB image is reserved by Guardian and inaccessible for the guest OS.

Secure Bootup. Figure 2 illustrates the disk layout for Guardian, where a special partition, referred to as the *hypervisor-partition*, is created during installation to avoid being trespassed by normal file systems. To allow for a secure boot without increasing the TCB size and complexity, we make slight changes on the bootloader (e.g., Grub 2). The BIOS passes the control to the bootloader core in the boot track. The bootloader core includes the Master Boot Record (MBR), the diskboot image and the basic-function image, which provides all basic functions and usually has to load other modules and configuration files such as *grub.cfg* to launch an operating system due to the limited size of the boot track (32KB in maximum).

Our modification is on the basic-function image only, such that it always launches Guardian *before* loading other components including the OS. In specific, once the core is loaded to the CPU by the BIOS (illustrated by Step 1 in Figure 3), it checks a bit flag in main memory (referred to as VMM_flag) which indicates Guardian's presence. If VMM_flag is not set, i.e., the core immediately passes the control to Guardian whose image is placed at a *fixed* disk address upon installation (Step 2 in Figure 3). The address of Guardian is hard-coded into the core, such that it loads Guardian directly using disk I/O without involving any file system.

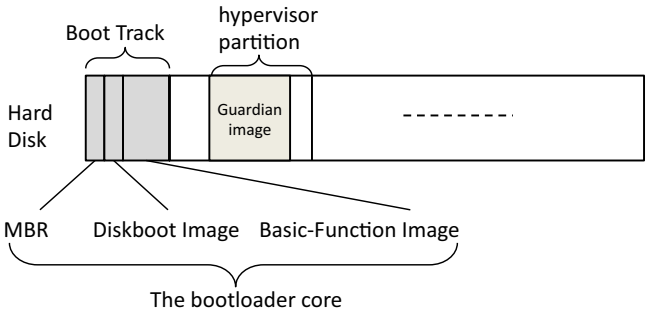

Fig. 2. An illustration of the disk layout

After occupying the CPU, Guardian loads the TCB image into a reserved memory region. It then configures the hypervisor page table, the EPT and IOMMU to ensure that the reserved region is not in the hypervisor or the guest's space and not accessible by DMA devices either. Separating the reserved region from the hypervisor space ensures no accidental accesses to the region. (As shown later, Guardian must map the region into its space by re-configuring the page table in order to access it.)

Finally, Guardian sets VMM_flag indicating its presence, and passes the control back to the bootloader core (Step 3). After asserting the flag is set, the core loads other modules and configuration files (Step 4) and proceeds to boot up the guest in the normal way (Step 5).

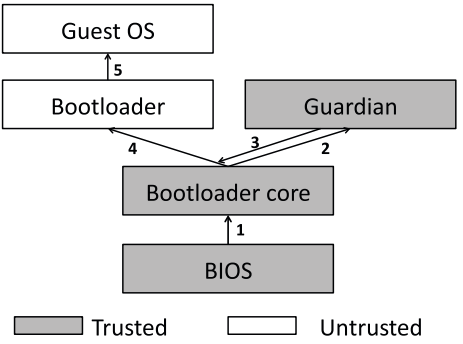

Fig. 3. The sequence of secure bootup

Device Configuration Space Protection. A rootkit may manipulate the device configuration space (e.g., the space-overlapping attack [44]) to thwart Guardian to intercept certain I/O events or access to I/O data. In order to defeat the configuration space manipulations and conflicts/overlapping between different devices, Guardian is poised to intercept and validate any update to the device configuration registers after its boot up. Note that these registers are located in the northbridge chipset [11]. The interception

are realized via configuring Virtual-Machine Control Structure (VMCS) for I/O ports and the EPT for MMIO regions.

Secure Shutdown. The guest may modify the Guardian image on the disk. Therefore, when the system is powered off, the TCB saved in the reserved memory must be written back to their original locations in the disk for the next round of execution. There exist two types of shutdown events. One type is the sleep events, where the system enters a sleep state through the Advanced Configuration and Power-management Interface (ACPI) [14]; the other is the reboot event, where the system restarts from the BIOS. Guardian intercepts both types of shutdown events and responds accordingly.

ACPI Sleep. The ACPI sleep event is managed by the Operating System Power Management (OSPM) subsystem on the modern ACPI-compatible system. Receiving commands from software (e.g., system call) or external interrupts (e.g., the System Control Interrupt triggered by pressing the power/sleep button or closing the laptop lid), the OSPM subsystem sets the PM1a_CNT register to force the system entering the corresponding sleep state. Note that Guardian prohibits the ACPI sleep event to be triggered by the optional sleep control and PM1b_CNT registers. Specifically, there is a 32-bit pointer in the Fixed ACPI Table (FADT) pointing to the PM1b_CNT block. Guardian clears this pointer and intercepts accesses to the PM1b_CNT register. The same method is used on the control sleep register.

Guardian intercepts the guest's sleep command issued to the PM1a_CNT register. Note that the actual interception method depends on whether the register is accessed by PIO or MMIO. The former involves VMCS configuration whereas the latter requires the EPT.

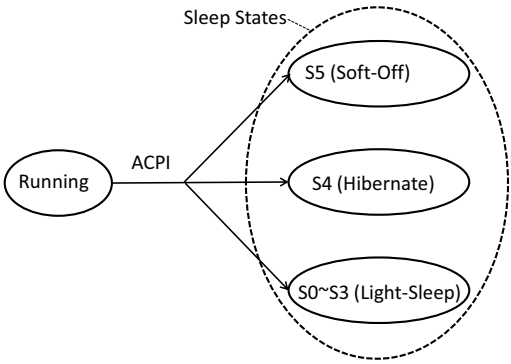

Fig. 4. ACPI sleep states

Among the six Sleep states ($S0$ to $S5$) defined in the ACPI specification (in Figure 4), the light-sleep ($S0$ to $S3$) states are not of concern, because the main memory remains powered and Guardian remains alive. Therefore, Guardian performs no action. For the soft-off state ($S5$) where the system will be powered off, Guardian restores the TCB

image back to the respective disk locations by using direct disk I/O operations. Note that Guardian needs to re-activate the disk which has been closed (but remains powered) before the ACIP sleep command is issued. In the end, Guardian clears VMM_flag and resumes the intercepted ACPI command which turns the platform off.

It is slightly more complicated to deal with the hibernation state $S4$ due to the need for platform context saving. Guardian needs to save its context into the hypervisor partition, in addition to the restoration work done for $S5$. For the guest context, Guardian disables and prohibits the ACPI S4BIOS Transition [1], which bypasses Guardian as the BIOS directly saves *all* memory content into the hard disk including Guardian's context. Therefore, only the OS-assisted hibernation method is supported and the OS must write its own context into the disk before hibernation.

Note that after the PM1a_CNT register is set, the platform passes the point of no return, because the ACPI hardware will force the platform to enter $S4$ or $S5$ state and no software will be loaded to the CPU. In other words, Guardian is the last piece of code executed before shutdown, which guarantees the security of the TCB and critical data resting on the disk.

System Reboot. There are three possible ways to reboot a system. One is ACPI reset, which is activated by the ACPI reset register. Note that the system will immediately reboot once the reset register is set. The ACPI reset register can be accessed by port I/O or memory-mapped I/O, which can be intercepted by Guardian through configuring the VMCS or EPT, respectively. The second way is essentially triggered by the CPU INIT signal. Guardian intercepts the event through configuring the VMCS.

In the third way, an attacker can switch the CPU to the real mode and jump to the BIOS entry to reboot the system. The tricky part is that it can bypass the INIT and ACPI reset mechanisms, meaning that the previous two interception methods will fail to intercept this one. To intercept it, a straightforward solution is to intercept the CPU switch from protected mode to real mode. However, the cost will significantly rise up when legitimate CPU-mode switches take place frequently, e.g., in Windows. Our solution is to prevent jumping to the BIOS reboot-routine from the guest by configuring the EPT. Any attempts from the guest OS to reboot the system will be intercepted by Guardian whose response is to repeat Step 3-5 in secure bootup without rebooting the whole platform.

Recovery. Guardian provides an alternative secure boot mechanism, where the system is able to boot up from a trusted-storage, such as a live CD or a read-only USB token. The bootup sequence is the same as the one described in Section 3.1. For convenience, the end-user can configure the system always boot up from a trusted storage, such that the system still can boot up into a trusted state.

The secure shutdown procedure may not be triggered due to some unexpected and irresistible events, e.g., power failure or system crash. Given that such unexpected system failure events may lead to the untrustworthiness of the TCB image, we need the TPM-based secure boot [37] to guarantee that only the trusted image can be booted.

[1] It clears the F bit in the Firmware ACPI Control Structure (FACS) and intercepts accesses to the SMI_CMD command register, which is S4BIOS service activation.

In such cases, the system can not boot up, and the security-conscious end-users need the recovery mechanism to restore Guardian image. Specifically, the bootloader in the trusted storage is extended to restore TCB image into the hard drive. Note that the bootloader originally has the capabilities to read/write the hard drive, the trusted storage and the main memory. Therefore, we can easily combine these functions to do the recovery.

3.2 Secure User-Hypervisor Interface

The secure interface is a duplex channel between the end-user and Guardian without involving the guest OS. Guardian shields the channel against any access from the guest. With the interface, the end-user can configure Guardian during its boot-up, and issue commands during runtime. For the sake of usability and simplicity, we do not rely on any external device such as a USB token. The user inputs are through the keyboard while the outputs are via the display in VGA mode.

Guardian provides two secure UIs. One is the Boot Up Secure User Interface (BUSUI), which is used in the secure boot phase before the guest starts to run. Since the platform then is in a trustworthy state, the implementation of BUSUI is straightforward. Guardian utilizes the BIOS services (i.e. INT 0x16 and 0x10) for input and output. The end-user activates it by holding a special key for a few seconds. In our current design, a user can deposit a text message to Guardian as a shared secret and can also input policies.

The other interface is the Run Time Secure User Interface (RTSUI), which is used after the guest boots up. The RTSUI can be dynamically launched by the end-user. RTSUI extends the secure user interface in KGuard [8]. Namely, Guardian securely receive inputs of a human user through a keyboard while it securely produces outputs through the display. Both the input and output paths are inaccessible to the guest OS. Since the interface in KGuard is only for password input, we extend it to a command-line interface such that the user can conveniently input commands and read responses.

4 Security Utilities

When designing security utilities based on Guardian, we endeavor to deal with threats plaguing normal end-users and system administrators. To this end, we propose a device monitor and a hyper-firewall.

4.1 Device Monitoring

A rootkit can misuse a peripheral device without the user's consent. For instance, it can quickly turn on the camera of a laptop to take a picture of the user and then turn it off. In a stealthy manner, it can also turn a network adaptor into the promiscuous mode so as to sniff the entire LAN traffic. We develop a Guardian utility to monitor the states of the camera and the network interface. In case of risky device usage, the end-user is alerted via the hypervisor-user interface or a beep sound. Note that the beep cannot be stopped by the adversary, because Guardian is able to intercept all accesses to that device.

Camera Control. Our design considers an external camera attached to the platform through a USB interface. (It can also be extended for a built-in camera.) The USB port

is controlled by an EHCI [16] or UHCI [15] controller. In either case, a *frame list*, with its base address specified by the *PERIODICLISTBASE* register, is used to queue I/O commands. To enable the camera, the driver must insert a *transfer descriptor* or TD to the frame list. The host controller automatically fetches it from the queue and responds properly.

Upon the user's activation command, the camera control utility makes use of the interception primitive to set read-only on the region for the base register, the frame list and the TD queue. If it detects a new TD with the open command `UVC_SET_CUR` for the camera, it alerts the user through a beep sound.

NIC Promiscuous Mode Control. The control on the network interface is simpler than EHCI. The Unicast Promiscuous Enabled (UPE) bit and the Multicast Promiscuous Enabled (MPE) in the Receive ConTroL Register (RCTL) are the flags that turns on the NIC's promiscuous mode. The monitoring utility intercepts the accesses to RCTL. Once the UPE bit or the MPE bit is set, an alert is raised to the user.

Note that Guardian and its utilities are not burdened with the complicated task of device management, for instance, to block illegal operations. This is to keep the hypervisor size small and more reliable.

4.2 Hyper-firewall

Recent attacks have shown that both application-level and OS-level firewalls can be disabled by rootkits. One solution proposed recently is the VMwall [32], which isolates the firewall in a separated domain (i.e., the Dom0 in the Xen setting). However, this approach dramatically increases the TCB size and requires the user to run two domains concurrently.

We propose in this section a more elegant and stronger solution called *hyper-firewall* as the firewall functions in the hypervisor space. The basic idea is that a Guardian utility interposes on network I/O. It drops illegal packets if their TCP/IP headers are not compliant to the firewall policies set by the end-user through the secure UI. Since Guardian does not comprise any NIC driver, this utility does not significantly increase Guardian's code size. The main challenge is how to intercept network packets in an efficient way. Before presenting the details, we briefly explain the network I/O mechanism.

The packet transmission mechanism is illustrated in Figure 5. The NIC makes use of a ring buffer (essentially a circular queue) to store *transmit descriptors* which point to the packets to transmit. The ring buffer has its base address saved in the TDBAL and TDBAH registers, has its size saved in the TDLENL and TDLENH registers, and has a head register and a tail register pointing to the queue head and tail respectively. The NIC always dequeues the descriptor pointed by the head register, and then fetches the corresponding packet. After retrieval, it advances the head pointer. The tail pointer is maintained by the device driver. To send a new packet, the driver enqueues one or multiple descriptors. Then, the tail pointer is also advanced. The NIC only uses the descriptors between the head and the tail. It stops transmission when the two pointers collide.

The packet receiving mechanism is analogous to the transmission mechanism. It also has a ring buffer storing *receive descriptors*, and has its own base address registers, length registers, and the head and tail registers. Initially, the driver allocates a set of fixed

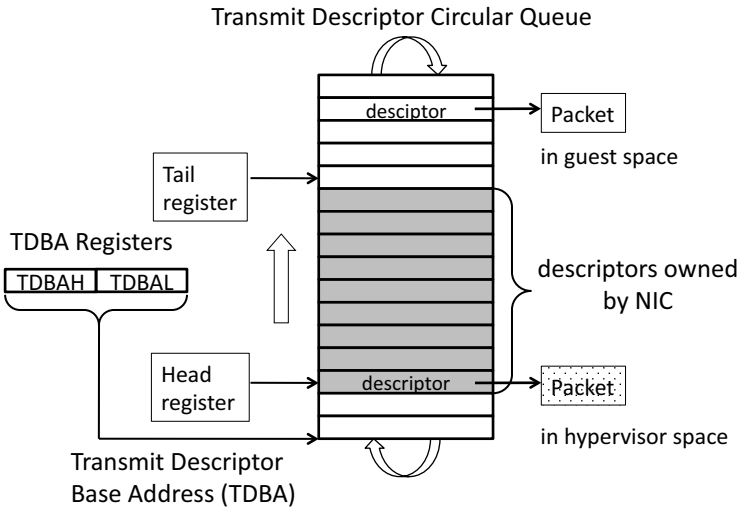

Fig. 5. The transmit descriptor circular queue used by the NIC

length DMA buffers, and enqueues the corresponding descriptors into the ring queue. When receiving packets, the NIC stores them into those pre-allocated DMA buffers, updates the corresponding descriptors, and advances the head pointer accordingly. Finally, it throws out an interrupt to notify the driver to fetch the packets according to the descriptors. Since the packet sending and receiving mechanisms are different, we design two interposition schemes, respectively. Note that the registers used by NICs may be different. To support all NICs, we can provide a profile which can provide necessary information for Guardian to understand register meanings.

Outbound Packet Filter. Guardian uses the EPT to intercept all write accesses the TDBAL, TDBAH, TDLENL and TDLENH registers so that Guardian can always locate the legitimate ring buffer. Similarly, it sets up the EPT and IOMMU tables, such that the head register can only be updated by the NIC[2], and all accesses to the tail register are intercepted by Guardian. Lastly, it sets the entire ring buffer as read-only.

When a write access to the ring buffer is intercepted by Guardian, it checks whether the write overwrites an existing descriptor which has not been fetched by the NIC. If so, the access is blocked; otherwise, Guardian emulates the write. When a write access to the tail register is intercepted, Guardian performs the following. (1) It checks whether the packets pointed by the descriptors between the present tail and the new tail are compliant with the firewall policies; (2) It copies all legal packets to the hypervisor space and updates those descriptors accordingly so that the NIC can fetch them from their new locations; for illegal packets, it sets the packet-length field in their descriptors as zero; (3) It emulates the tail update.

[2] In the current hardware specification, the driver is not able to instruct the NIC to update the header register.

Once the packets are moved to the hypervisor space, their descriptors are not allowed to be changed. Note that packets are much smaller than a memory page. Therefore, relocating them into the hypervisor space avoids undesirable page faults as compared to protecting them in the guest space.

Inbound Packet Filter. The inbound packet filter mechanism is similar to its outbound counterpart. By enforcing access control on those control registers and the ring buffer for the receiving descriptor, Guardian locates the DMA buffers allocated by the driver. To retrieve a packet, the driver first fetches the receive descriptor which triggers a page fault. Guardian then performs the packet inspection according to the firewall policies, and drops illegal ones.

5 Implementation

We have built a prototype of Guardian on a Dell OptiPlex 990 MT desktop with an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz processor[3] and 4GB main memory. Guardian consists of around 25K SLOC for its core functions, which is much smaller than Xen (263K SLOC for Xen-4.1.2) and Linux (8,143k SLOC for Linux-2.6.33.20). A comprehensive comparison between Guardian and other hypervisors is listed in Figure 6.

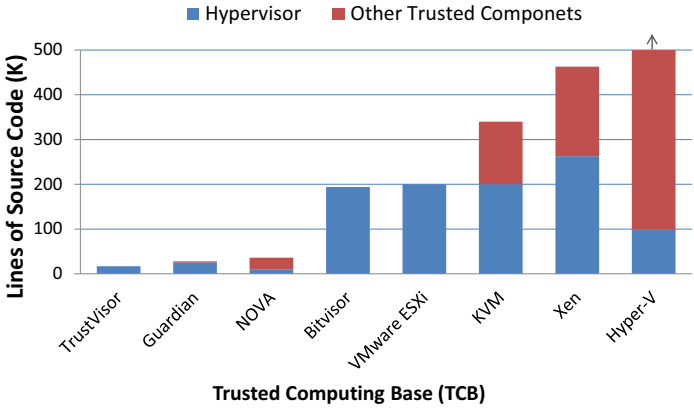

Fig. 6. Comparison of the TCB size. TrustVisor itself is around 17K SLOC. NOVA [33] consists of the microhypervisor (9K SLOC), and several trusted components, i.e., a thin user level environment (7K SLOC), and the VMM (20K SLOC). BitVisor [31] and VMware ESXi are 194K and 200K SLOC, respectively. KVM is around 200K SLOC, as well as a customized QEMU (140K SLOC). Xen is around 263K SLOC with Dom0 that can be customized to 200K SLOC [19]. Microsoft Hyper-V uses a Xen-like architecture with a hypervisor (around 100K SLOC) and Windows Server 2008 (larger than 400K SLOC).

[3] The Hyper-threading mode is disabled since our current hypervisor does not support the multiprocessor mechanism.

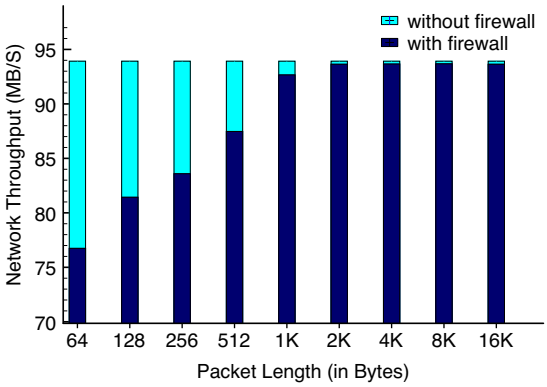

Fig. 7. The benchmark results with and without hyper-firewall

The binary size of Guardian is around $223KB$, which is much smaller than Xen (around 1,264KB for Xen-4.1.2) and Linux (around 134,134KB for Linux-2.6.33.20) image, and the bootloader core is around $30KB$. Guardian reserves 512KB memory space for TCB images and other critical information. Guardian also provides 11 hypercalls for security services, which is smaller than Xen exported hypercall surfaces (i.e., 46 hypercalls). Note that Guardian only focus on the security services, while these systems (e.g., Xen) usually provide many more functional services.

5.1 Device Monitoring Evaluation

The device management component consists of 1.2K SLOC. Currently Guardian supports to monitor camera and network card working modes. It can be extended to support other similar devices, such as a microphone.

We experiment with a USB Logitech web camera attached on an EHCI host controller. Note that the monitoring has no effect on the camera's performance as the scheme does not intercept runtime commands and data transferring.

The network card mode monitor is built upon the Intel 82579LM Gigabit Network Card, whose registers are accessed using MMIO. The experiment results produced by network benchmark tool *netperf* [25] prove that the monitor service almost does not affect the network I/O throughout. Note that the device management service does not require any modifications in the guest kernel or device drivers.

5.2 Hyper-firewall Evaluation

The packet filter service is built on the Intel Corporation 82579LM Gigabit Network Card, and does not add any code into the guest OS. Current hyper-firewall supports adding policies on inbound and outbound packets. For the outbound packets, hyper-firewall restricts the region of the target destination (e.g., external IP addresses), and for the inbound packets, hyper-firewall restricts the connection ports (e.g., SSH port 22).

All hyper-firewall policies can be enabled and disabled through the RTSUI. All experiments show the hyper-firewall works well. We tested the network I/O performance with benchmark tool *netperf* [25]. When we only enable outbound policy, the performance results show that our hyper-firewall only introduces (0.096% - 0.064%) performance overhead; when we enable inbound and outbound policies, the hyper-firewall introduces (18.29% - 0.26%) performance overhead. Note that the short packet setting generates more interceptions. Thus its performance is relatively low. Note that the monitoring of NIC does not affect the I/O speed of other derives. The packet filter service only adds $0.9K$ SLOC into Guardian.

5.3 System Benchmark

We first measure the overhead on the OS operations using the LmBench suite. Figure 8 shows the results: socket (local connection), memory operations (i.e., read, write and bcopy) and some system calls (i.e., mmap, fork+exec and fork+exit). However, fork+exec and fork+exit incur higher performance penalties of 39% and 38%, which are heavily dependent on the Intel EPT performance. We do believe that this could be improved with the performance enhancing of memory virtualization.

We also measure computation performance with Guardian. The results generated by the benchmark tool SPEC CPU 2006 (see Figure 9) show that Guardian usually only introduces 0.2% - 10.3% performance loss, and may lead to 38.2% performance overhead in some extreme cases (i.e., memory intensive operations with extreme low cache hit rate), which is also dependent on the page operations of current Intel EPT. Again, we believe that it can be improved in the further.

For I/O-bound benchmark test, we select a range of benchmark tools, including Bonnie, Postmark, netperf and Linux kernel. For Bonnie, we use a 1GB file and perform sequential read/write (fread/fwrite) and random access (frandom). For Postmark, we choose 20,000 files, 100,000 transactions and 100 subdirectories, as well as all other default parameters. For netperf, we use another local machine as the netperf server, and

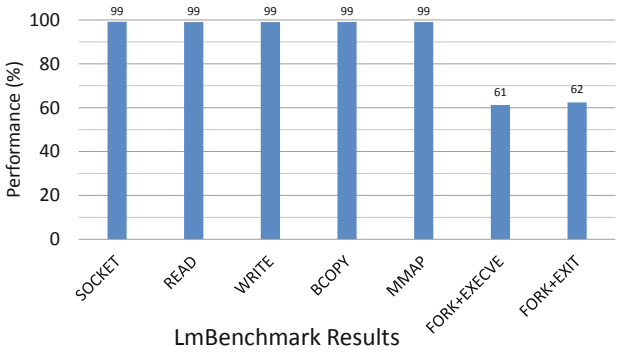

Fig. 8. The LmBench results on OS operations

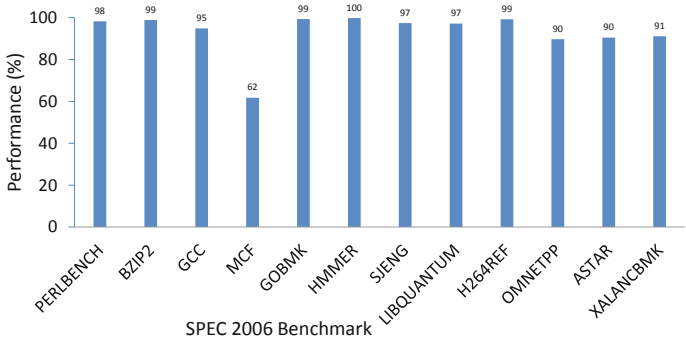

Fig. 9. The system benchmark comparison results generated by *SPEC CPU 2006*

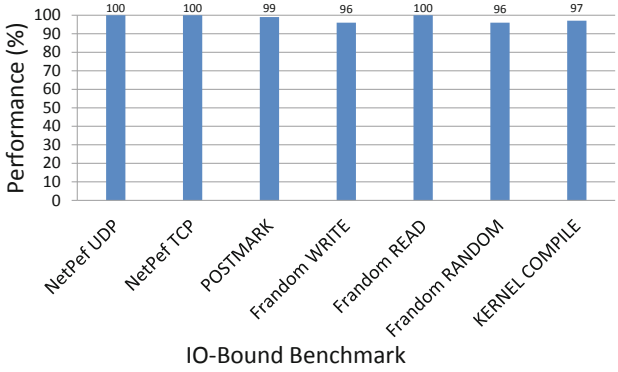

Fig. 10. The I/O-bound benchmark results

run both TCP STREAM and UDP STREAM benchmarks to measure basic network performance. For Linux kernel, we compile the Linux-2.6.33.20 with default configuration. Figure 10 shows the results.

6 Related Work

Software-Based Root of Trust. Software-based ROTs have been proposed and used in [27,29,30]. The trust establishment is based on a challenge-response protocol. A speed-optimized function (code block) is established as the ROT on a platform if, within an acceptable time delay, it can compute a correct checksum of memory regions according to a given challenge. It is based on the assumption that it incurs a noticeably longer delay for any other implementation of this function. It also has a restriction on both the adversary's capability, for instance no collusion with a third party, as mentioned in [10]) and the capabilities of the target platforms. In addition, to stop the proxy attack, it may even require to unplug the network and disable the wireless to physically cut down the connection with outside. These limitations and requirements lead to inconvenience or even to impracticability. Thus, software ROTs are unqualified to be a security foothold for normal users' computers.

Hardware-Based Root of Trust. The hardware-based ROT can be categorized into static ROTs and dynamic ROTs. A static ROT is a built-in platform component. When the platform boots up, a trust chain can be established from the ROT up to the operating system. The TPM chip [37] is a typical example of static hardware ROT. As a chip on the motherboard, it is secure against all software attacks. Secure (or authenticated) boot up, remote attestation and sealed storage are the main security services provided by the TPM framework. The main disadvantages of TPM are its low speed, inflexibility and passiveness. Therefore, to support various security services, it usually requires assistance from certain secure software routine (e.g., hypervisor). IBM's secure co-processor [2] is a strong hardware root of trust with such a high price tag that it is not feasible for the mass market. SMART [10] is a hardware-software co-designed scheme, where a piece of code works on a modified *low-end* microcontroller units (MCU) to function as a dynamic ROT. The SwitchBlade architecture [5] can prevent persistent rootkits from infecting security-critical files (e.g., kernel image) with an ROT residing on the disk controller. These ROTs may be integrated with Guardian though carefully design and implementation.

AMD Secure Virtual Machine (SVM) [1] and Intel Trusted Execution Technology (TXT) [18] are dynamic ROTs. These new processor features allow a piece of code to be securely executed in an isolated environment enforced by the hardware. Despite of their easiness of use, they incur high latency as showed in the Flicker system [22]. Fortunately, the high latency may be tolerable for the end-users, since it only required once when the system as well as Guardian boots up. The boot mechanism of Guardian is compatible with dynamic ROT techniques.

Hypervisor Related Security Systems. Many hypervisor-based security systems have been designed and reported in the literature. For instance, a hypervisor can be applied for I/O related protection [9,31], for kernel integrity protection [3,13,23,26,28,41,42], and for user space protection [6,7,12,21,34,43]. By studying these systems, we identify cryptographic engine, measurement, emulation, interception and manipulation as the fundamental security primitives which are adopted in Guardian as well.

Our work has remarkable differences with the aforementioned systems. Guardian aims to be a versatile hypervisor. By bring together a number of fundamental security primitives, Guardian facilitates the design and implementation of virtualization-based security systems, rather than focusing only on a single security problem. In addition, Guardian caters to the security needs of the end-user, which demands Guardian to be highly efficient, easy-to-use and compatible with the operating system and applications

Note that those schemes [4,24,39,40] that enhance the hypervisor security are complimentary to our work. The security of Guardian will be further improved if these techniques are applied in its implementation.

7 Conclusion

In this paper, we have proposed Guardian as a security foothold on the end-user systems to enhance their security. Specifically, we introduced Guardian whose integrity and availability were guaranteed by the novel bootup and shutdown technique. Guardian also provided a secure user interface, through which the end-user could update the

configurations of Guardian or dynamically activate/deactivate a dedicated security service for the security needs. We also proposed two security utilities based on Guardian: a device monitor which detects malicious device operations and a hyper-firewall which inspects the incoming and outgoing network packets from the hypervisor space. We have implemented Guardian and the two utilities. The experiment results show that they are efficient and easy to use. Our work demonstrates that computer security can be significantly boosted up by using a tiny and reliable hypervisor.

Acknowledgement. We thank the reviewers and especially Mohammad Mannan for their constructive comments. This research/project is supported by the Singapore National Research Foundation under its International Research Centre@ Singapore Funding Initiative and administered by the IDM Programme Office.

References

1. AMD. Secure virtual machine architecture reference manual. Technical report (2005)
2. Arnold, T.W., Van Doom, L.P.: The IBM PCIXCC: a new cryptographic coprocessor for the IBM eserver. IBM J. Res. Dev. 48(3-4), 475–487 (2004)
3. Azab, A.M., Ning, P., Sezer, E.C., Zhang, X.: HIMA: A hypervisor-based integrity measurement agent. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009, pp. 461–470. IEEE Computer Society, Washington, DC (2009)
4. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 38–49. ACM, New York (2010)
5. Butler, K.R.B., McLaughlin, S., Moyer, T., McDaniel, P.D.: New security architectures based on emerging disk functionality. IEEE Security and Privacy Magazine (September 2010)
6. Champagne, D., Lee, R.B.: Scalable architectural support for trusted software. In: Jacob, M.T., Das, C.R., Bose, P. (eds.) HPCA, pp. 1–12. IEEE Computer Society (2010)
7. Chen, X., Garfinkel, T., Christopher Lewis, E., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA (March 2008)
8. Cheng, Y., Ding, X.: Virtualization based password protection against malware in untrusted operating systems. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) Trust 2012. LNCS, vol. 7344, pp. 201–218. Springer, Heidelberg (2012)
9. Cheng, Y., Ding, X., Deng, R.H.: Driverguard: a fine-grained protection on I/O flows. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 227–244. Springer, Heidelberg (2011)
10. Eldefrawy, K., Francillon, A., Perito, D., Tsudik, G.: SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium, San Diego, USA, February 5-8 (2012)
11. Fleming, S.: Accessing pci express configuration registers using intel chipsets. otechnical report (2008)
12. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. In: Proceedings of the 9th ACM Symposium on Operating Systems Principles, pp. 193–206. ACM, New York (2003)

13. Grace, M., Wang, Z., Srinivasan, D., Li, J., Jiang, X., Liang, Z., Liakh, S.: Transparent protection of commodity OS kernels using hardware virtualization. In: Jajodia, S., Zhou, J. (eds.) SecureComm 2010. LNICST, vol. 50, pp. 162–180. Springer, Heidelberg (2010)
14. Hewleet-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. (Revision 3.0b) (October 2006)
15. Intel. Universal host controller interface (UHCI) design guide (March 1996)
16. Intel. Enhanced host controller interface specification for universal serial bus (March 2002)
17. Intel. Intel I/O controller hub 9 (ICH9) family datasheet (2008)
18. Intel. Intel Trusted Execution Technology (Intel TXT) software development guide (December 2009)
19. Keller, E., Szefer, J., Rexford, J., Lee, R.B.: Nohype: virtualized cloud infrastructure without the virtualization. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA 2010, pp. 350–361. ACM, New York (2010)
20. Li, Y., McCune, J.M., Perrig, A.: Viper: verifying the integrity of peripherals' firmware. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 3–16. ACM, New York (2011)
21. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: Trustvisor: Efficient TCB reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 143–158. IEEE Computer Society, Washington, DC (2010)
22. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the ACM European Conference in Computer Systems (EuroSys) (April 2008)
23. de Oliveira, D.A.S., Felix Wu, S.: Protecting kernel code and data with a virtualization-aware collaborative operating system. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009, pp. 451–460. IEEE Computer Society, Washington, DC (2009)
24. Rafal, W., Joanna, R., Alexander, T.: Xen 0wning trilogy, Black Hat conference (2008)
25. Rick, J.: Network Performance Benchmark Tool - Netpref, `http://www.netperf.org/netperf/`
26. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
27. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: Scuba: Secure code update by attestation in sensor networks. In: Proceedings of the 5th ACM Workshop on Wireless Security, WiSe 2006, pp. 85–94. ACM, New York (2006)
28. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 335–350. ACM, New York (2007)
29. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 1–16. ACM, New York (2005)
30. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.K.: SWATT: Software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy (2004)
31. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., Kato, K.: Bitvisor: a thin hypervisor for enforcing I/O device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2009, pp. 121–130. ACM, New York (2009)

32. Srivastava, A., Giffin, J.: Tamper-resistant, application-aware blocking of malicious network connections. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 39–58. Springer, Heidelberg (2008)
33. Steinberg, U., Kauer, B.: Nova: A microhypervisor-based secure virtualization architecture. In: Proceedings of the European Conference on Computer Systems (2010)
34. Strackx, R., Piessens, F.: Fides: selectively hardening software application components against kernel-level or process-level malware. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, pp. 2–13. ACM, New York (2012)
35. Sun, K., Wang, J., Zhang, F., Stavrou, A.: SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium, San Diego, California, USA (2012)
36. Phoenix Technologies: Trustedcore: Foundation for secure CRTM and BIOS implementation (2006), `https://forms.phoenix.com/whitepaperdownload-/docs/trustedcore_wp.pdf`
37. Trusted Computing Group: TPM main specification. Main Specification Version 1.2 rev. 85 (February 2005)
38. Vasudevan, A., Parno, B., Qu, N., Gligor, V.D., Perrig, A.: Lockdown: A safe and practical environment for security applications (CMU-Cylab-09-011) (2009)
39. Wang, J., Stavrou, A., Ghosh, A.: HyperCheck: A hardware-assisted integrity monitor. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 158–177. Springer, Heidelberg (2010)
40. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 380–395. IEEE Computer Society, Washington, DC (2010)
41. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
42. Xiong, X., Tian, D., Liu, P.: Practical protection of kernel integrity for commodity os from untrusted extensions. NDSS (2011)
43. Yang, J., Shin, K.G.: Using hypervisor to provide data secrecy for user applications on a per-page basis. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2008, pp. 71–80. ACM, New York (2008)
44. Zhou, Z., Gligor, V.D., Newsome, J., McCune, J.M.: Building verifiable trusted path on commodity x86 computers. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2012)