# Shadow-Box: The Practical and Omnipotent Sandbox


Seunghun Han
National Security Research Institute
hanseunghun@nsr.re.kr

Junghwan Kang
National Security Research Institute
ultract@nsr.re.kr

Wook Shin
National Security Research Institute
wshin@nsr.re.kr

HyoungChun Kim
National Security Research Institute
khche@nsr.re.kr

Eungki Park
National Security Research Institute
ekpark@nsr.re.kr



## Abstract

We propose a security monitoring framework for operating systems, Shadow-box, exploiting state-of-the-art virtualization technologies. Shadow-box is primarily composed of a lightweight hypervisor and a security monitor. As being loaded on an operating system, the lightweight hypervisor prepares a guest machine and project the operating system upon the guest. The hypervisor partially shadows the guest kernel for the purpose of security investigation and recovery, and runs the security monitor to supervise accesses and sanity of the guest. We manipulate address translations from the guest to the physical memory in order to exclude unauthorized accesses to the host and the hypervisor spaces. In that way, Shadow-box can properly introspect the guest operating system and mediate all accesses, even when the operating system is compromised. The Shadow-box could be used for various security enforcements, such as malware filtering, information flow control, auditing, etc. We exemplify the security monitor by implementing an integrity protector for an operating system kernel, and show how it effectively neutralizes rootkits and malicious root accesses of malware in Linux and Android. Performance evaluation results are presented as well. Shadow-box provides stronger protections with less overhead than existing hypervisor-based security solutions. Moreover, all the protections are applied to an existing system on-the-fly. The hypervisor does not have to be installed antecedently.


## 1. Introduction

Rootkits are clandestine malware capable of obtaining administrator privileges and manipulating kernel objects. The kernel objects include a wide variety of codes and data, such as kernel texts, modules, function tables, process lists, page tables, etc. Rootkits play an important part in practical cyber-attacks; Relying on tunnels provided by a lurking rootkit, other malware sneaks in and subverts the kernel of the fortress. McAfee, Inc. [3, 4] reports that the number of rootkits has been increased by 33% over the past three years and kept increasing.

Protection rings [11] define hierarchically ordered domains where privileges for accessing system resources are differentiated. An access across the rings is restricted. Both Windows and Linux operating systems implement two levels of the rings: *Ring 0* and *Ring 3* for running the kernel and user applications, respectively. It had been believed that defense mechanisms better be placed at *Ring 0* for regulating malware at *Ring 3*, however the belief becomes obsolete due to proliferation of rootkits. Rootkits can escalate their privilege levels, modify kernel objects, and even disable anti-malware solutions.

Some anti-rootkit solutions [19, 25, 30, 34, 37, 39] tackle problems by employing virtualization technologies. Recent CPU vendors even provide hardware instructions to support virtualizations and create a new privilege level underneath *Ring 0*. The new level, *Ring -1*, offers a set of instructions controlling *Ring 0* accesses. Those instructions can be utilized by a hypervisor, a piece of software that enables a physical machine to run multiple heterogeneous systems exclusively on virtual execution environments. Namely, anti-rootkit solutions running at the hypervisor level would remain unharmed and sustain their functions even when guest virtual machines (VMs) are compromised.

Downsides of hypervisor-based security solutions are performance overheads and semantic inconsistency due to virtual state managements. Hypervisor processes, like security monitors, need to introspect VMs for figuring out their status and behavior. Virtual machine state information is accessible when the hypervisor reads raw-bits from virtual memory, virtual registers, and virtual devices, however, interpretation of the raw-bits requires such reconstruction processes as address translations, symbol resolutions,


This paper was presented at *HITBSecConf 2017*, April 13–14, 2017, Amsterdam, Netherlands
An earlier version of this paper was presented at *Black Hat Asia 2017*, "Myth and Truth about Hypervisor-Based Kernel Protector: The Reason Why You Need Shadow-Box"

| Name | Modified Kernel Object | Type | Attribute | Note |
|---|---|---|---|---|
| EnyeLKM 1.3 | syscall_trace_entry | Code | Static | code change, |
| | sysenter_entry | Code | Static | syscall hook, |
| | module->list | Data | Dynamic | direct kernel object |
| | init_net->proc_net->subdir->tcp_data->tcp4_seq_show | Function pointer | Dynamic | manipulation (DKOM) |
| Adore-ng 0.56 | vfs_root->f_op->write | Function pointer | Dynamic | function pointer hook |
| | vfs_root->f_op->readdir | Function pointer | Dynamic | |
| | vfs_proc->f_dentry->d_inode->i_op->lookup | Function pointer | Dynamic | |
| | socket_udp->ops->recvmsg | Function pointer | Dynamic | |
| Sebek 2.0 | sys_call_table | System table | Static | syscall hook, |
| | vfs_proc_net_dev->get_info | Function pointer | Dynamic | function pointer hook, |
| | vfs_proc_net_packet->proc_fops | Function pointer | Dynamic | DKOM |
| | module->list | Data | Dynamic | |
| Suckit 2.0 | idt_table | System table | Static | idt hook, |
| | sys_call_table | System table | Static | syscall hook |
| kbeast v1 | sys_call_table | System table | Static | syscall hook, |
| | init_net->proc_net->subdir->tcp_data->tcp4_seq_show | Function pointer | Dynamic | function pointer hook, |
| | module->list | Data | Dynamic | DKOM |

**Table 1.** List of well-known Linux kernel rootkit and modified kernel objects

record reconstructions, etc. Namely, it may cause performance overhead and inconsistency between semantic views of the hypervisor and VMs. VM introspection may decrease portability as well because it depends on implementation details of guest operating systems.

There are diverse approaches of reducing virtualization overheads and semantic gaps. SecVisor [34] employed a lightweight hypervisor and offered protection for static kernel objects such as kernel codes and read-only data [25, 30]. Others used hardware devices dedicated for security [23, 24, 26, 41]. Auxiliary hardware cannot lessen performance burden only, but also help guarantee the integrity of security-related procedures engraved in circuits. Malware may overthrow guest operating systems, but cannot compromise hardware logic. Dedicated hardware, however, increases costs and decreases the portability, and there is still semantic gap between the context of the CPU and the dedicated one.

In this paper, the hypervisor-based approach is revisited. We designed and implemented a lightweight hypervisor which facilitates security enforcements at *Ring -1* level. By making the host and the guest operating systems share most parts of the kernel space, our hypervisor works with negligible performance cost and semantic gap. The hypervisor is a sort of type 2, designed as a loadable kernel module (LKM), and can be applied to an existing system. Based on CPU and I/O virtualization supports, we design *Shadow-box*, a hypervisor-based monitoring framework that supports periodic and event-based monitoring on kernel objects. Benefits of Shadow-box are exemplified by implementing an integrity monitor that recognizes integrity breach in static and dynamic kernel objects. We show how it can rule out system attacks effectively by running five well-known rootkits on a machine.

We measured performance overheads from the proposed system by running benchmarking tools in single-core and multi-core processor settings. To compile a Linux kernel, the proposed system imposes about 5.3% of overheads in the single-core processor setting and 6.2% of overheads in the multi-core processor setting.

Our contributions can be summarized as follows;

- We present a security enforcing framework, Shadow-box, based on a lightweight hypervisor. It comes with less virtualization overheads and semantic gap than existing hypervisor-based security solutions. Shadow-box does not require kernel modification or patches, and hence it can be applied to existing systems without re-installation.
- We propose a practical countermeasure against rootkits using Shadow-box. After looking into well-known rootkits and related work, we classified static and dynamic kernel objects that are altered by the rootkits. Shadow-box tests and guarantees integrity of the classified kernel objects. The classification results can be used by other anti-malware solutions as well.
- We design event-based and periodic monitoring interfaces of Shadow-box considering various security needs. Other than a kernel integrity monitor, diverse security-related applications such as an auditor, an intrusion detector, and a security assessment tool can be built upon Shadow-box to achieve their own goals with hypervisor-level privileges.

## 2. Background

### 2.1 Rootkit

A rootkit is malware that is characterized by its nature of hiding and privilege escalation. Once after a rootkit successfully elevated its privilege level to administrator's, it often forges variety of kernel objects including internal data structures for managing processes, files, and modules for the purpose of persistent attacks. It may place a backdoor and lead to influx of other malware. It also may establish a hidden communication channel with a remote attacker.

**Rootkit Categories:** Rootkits are two sorts. User-level rootkits run with user privileges. They forge system commands such as `ls`, `ps`, `netstat`, etc. They also may implant hooks [40] to falsify system information and lurk in the system using linker preload directives to replace core libraries such as libc. Kernel-level rootkits, with administrator privileges, are capable of altering kernel objects such as kernel texts, function pointers, system call tables, and interrupt descriptor table (IDT) [38], that are normally out of reach of users. Since kernel-level rootkits are even able to neutralize kernel-level anti-malware solutions, the rootkits have become prevalent and drawn significant attention.

**Kernel-level Rootkits and Kernel Objects:** In Table 1, we enumerated the kernel objects that are frequently tampered by well-known rootkits [14, 23, 28, 29], which again can be categorized into static and dynamic kernel objects.

*Static Kernel Objects:* The static objects that reside in read-only memory area include kernel codes (texts), the system call table (`sys_call_table`), and the interrupt descriptor table (`idt_table`). Read-only data of loadable kernel modules also fall into this class. A rootkit can set the static kernel objects writable, and then alter the objects by registering codes, system calls, and interrupt handlers.

*Dynamic Kernel Objects:* The dynamic objects reside in writable memory area. The list of all processes and installed kernel modules, which can be traversed by `init_task` and `module`, are dynamic kernel objects. Files and sockets are also dynamic ones as they are located in the kernel heap and store mutable values. File and socket objects define available operations in their structure. The implementation instances of the operations, which are function pointers, are normally in read-only memory area. However, an attacker can set the memory area writable, besides it is possible to replace the pointers toward the operation instances with bogus pointers toward malicious codes.

### 2.2 Virtualization Technology

Hypervisor, also known as virtual machine monitor (VMM), allows a host physical machine (or the host) to run multiple guest virtual machines (or the guests). Hypervisor virtualizes computing resources of the host, such as CPU, main memory, storage, and network, so that the guests share the abstracted resources and run independently. Thanks to the abstraction, multiple guests can even run different operating systems.

Hypervisor can be categorized into two types [27]; Type 1 hypervisors (or bare-metal hypervisors) are installed and run on host hardware directly, whereas Type 2 hypervisors require operating systems installed beforehand on the host. That is to say, a Type 2 hypervisor is a sort of an application program. Xen [15], VMware ESXi [1], and Microsoft Hyper-V [2] fall into the Type 1, while KVM [21], VMware Workstation [7], and Oracle VirtualBox [6] do into the Type 2. Recent hypervisors do not come with software stack only. Modern processors are equipped with hardware-assisted virtualization (HAV) technologies, such as Intel VT-x [11], AMD-V [9], and ARM TrustZone [5], for better performance. Complexities and overheads from CPU-, memory-, and I/O-virtualization are reduced by leveraging HAV.

## 3. Assumptions

Our CPU is supposed to be equipped with virtualization technologies (VT) such as Intel VT-x [11] and AMD-V [9]. The main board chipset is also assumed to have I/O virtualization supports such as Intel VT-d [8] and AMD-Vi [10]. A system is assumed to be booted properly, utilizing the existing secure booting methods, such as *secure boot* [13], *verified boot* [18], and *tboot* [12]. The secure booting process should guarantee the integrity of the bootloader, the kernel, and loadable kernel modules including Shadow-box. After Shadow-box is loaded correctly, it can defeat the recent attacks against bootstrapping, such as BIOS [17], UEFI [32], and bootloader [22].

Attackers are omnipotent in this paper, after the Shadow-box is loaded. There is no limit for an attacker to install and run rootkits and other malware on the system. Attackers can even load their own kernel modules, alter memory via direct memory access (DMA), and attach any peripheral devices. Attackers can monitor, filter, and change any code and data of user, system, and hypervisor, in order to steal valuable information or compromise the system. We, however, exclude the cases of abusing resources only for reducing the availability. Denial-of-service, such as repeated rebooting or storage wiping out, is not our concern here.

## 4. Design

We explain how we designed the Light-box and the Shadow-watcher. It is designed to support a lightweight and practical security monitoring framework using virtualization technologies.

### 4.1 Light-Box Design

We developed a security monitoring framework, Shadow-box that keeps an OS safe by filtering out unauthorized accesses to important kernel elements and defending integrity of kernel elements periodically. Shadow-box relies upon its two sub-parts: a lightweight hypervisor and a security monitor. The lightweight hypervisor, *Light-box*, efficiently isolates an OS inside a guest machine, and projects static and dynamic kernel objects of the guest into the host machine, so that our security monitor in the host can investigate the projected images. The security monitor, *Shadow-watcher*, places event monitors on static kernel elements and tests se-

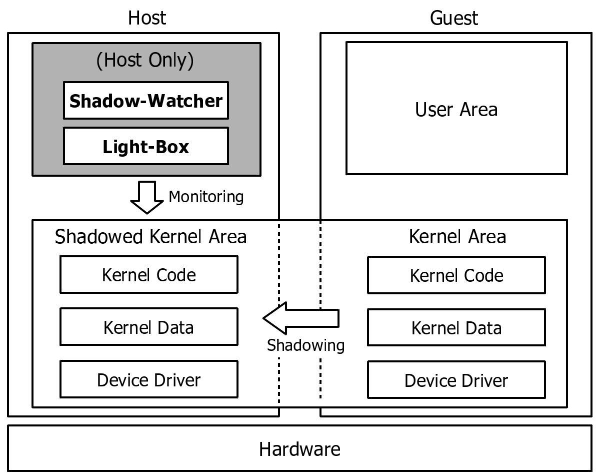


**Figure 1.** Block architecture of Shadow-box

curity of dynamic kernel elements. Running inside the host, it can test the security of the guest without malicious interference even when the guest OS is compromised.

Light-box does not require another OS underneath, which means it could be considered to be a sort of type 2 hypervisor [6, 7, 21]. Neither requires it a privileged virtual machine instance as Xen [15] does. Light-box can virtualize an installed OS, while many of existing hypervisor-based protections require installation of a hypervisor prior to the OS that needs to be protected. We implemented Light-box to be as compact as possible to minimize virtualization overheads and maximize available resources for the guest.

Figure 1 shows the design of Shadow-box. Light-box separates the guest and the host and shadows important kernel information on the host area. The host runs monitoring logics to guarantee security of the guest machine. Light-box is designed for providing an isolated execution environment for the purpose of protecting an operating system from system attacks, not for instantiating multiple virtual machines.

Consequently, the hypervisor better be as compact as possible to minimize virtualization overheads and maximize resource allocations for the guest. Light-box lets the host and the guest machines share the kernel space as shown in Figure 1. It promises less virtualization overheads. Moreover, it also reduces the semantic gap between the host and the guest. The less we have semantic gap between the host and the guest the more accurate and efficient monitoring logic on the guest behavior we have.

The share between the host and the guest, however, can create security weaknesses. Compromising the kernel space of the guest immediately affects the host, and it may lead to the subversion of the hypervisor. Light-box precludes any improper access to the shared area based on the following building blocks;

- **Security Bootstrapping**: It has to be guaranteed that important parts of Shadow-box are not corrupted by the time of system booting. *Security Bootstrapping* is to provide initial protections for the important parts by employing an existing secure booting technique.
- **Memory Separation and Protection**: The memory space of security bootstrapping has to be separated from the guest machine. After the system is booted, unauthorized accesses to the separated area are not possible because the area is protected from the guest.
- **Privileged Register Protection**: Some registers, called *privileged registers*, are important to manage execution of the guest OS and to maintain different levels of privileges. Unauthorized accesses on the registers are prohibited.
- **OS Independent Execution Flow**: We generate an independent control flow, rather than using the main kernel control flow. The independent one can be used for periodic testing for various purposes, such as integrity monitoring.

The above is described further in the following subsections.

### 4.1.1 Security Bootstrapping

At the moment of Shadow-box being loaded, the shared kernel area between the host and the guest is supposed to be clean. Important kernel objects and procedures, including our monitoring logic, reside in the kernel area. Disruption of the area brings compromising of the whole system. By taking advantage of existing secure booting supports, such *secure boot*, *verified boot*, and *tboot*, we can guarantee the integrity of the kernel at the stage of system booting. We put important loadable kernel modules in the kernel area, ahead of Shadow-box. Shadow-box ought to protect those preloaded modules after all. We do not maintain the whitelist which keeps the names of trustworthy modules, differently from NICKLE [30] and Lares [25], and get free from burden of managing the list.

### 4.1.2 Memory Separation and Protection

The accesses of shared kernel in the guest need to be controlled for protecting the host. Especially, the code and data of Shadow-box has to be out of reach for the guest. We keep the important codes being separated from the guests exploiting the page mapping data structures of the system. We maintain shadowed data that the Shadow-watcher uses as fresh as possible so that it can correctly reflect the execution status of the guest machine. These processes are explained further below:

**On-access Shadowing:** Light-box uses the page tables not only for the purpose of address translation, but also for protecting memory spaces. Page tables convert logical addresses that the guest machine uses into the physical addresses that the host machine manages. A hypervisor needs to update page tables to preserve consistent execution context between the host machine and guest machines. Light-

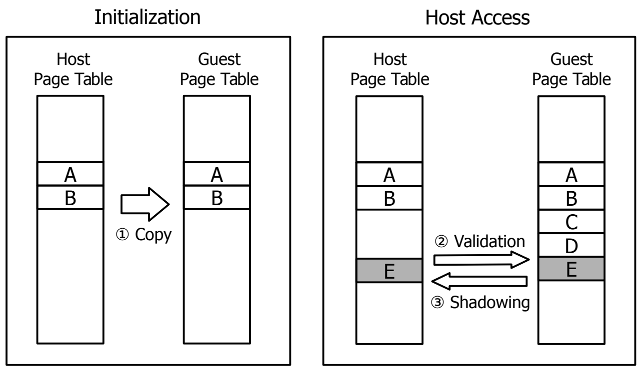


**Figure 2.** Operation of on-access shadowing

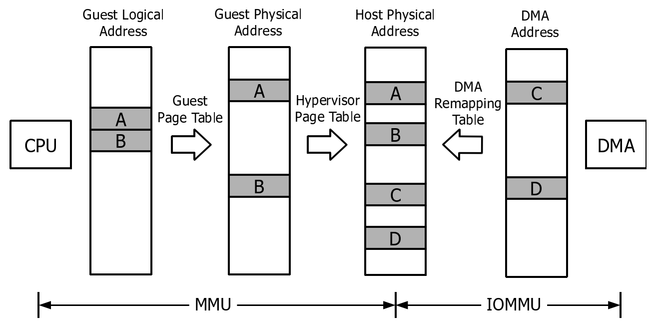


**Figure 3.** Address translation of CPU and DMA

box synchronizes page tables in an on-the-fly manner. It identifies when and where the host machine looks into the guest machine's memory space, and selectively synchronizes only the accessed area. Figure 2 shows how the tables are managed; when Light-box starts, it makes copy of the page table in which the security bootstrapping information is included. The duplicated page table is called *shadow page table*, and only accessed by the host machine. If the host needs to access the memory of the guest, the hypervisor checks the validity of guest's page table. If it has valid information, the host's table is synchronized with the guest's.

This way of page table synchronization, which we call *on-access shadowing*, imposes lower overheads than the other hypervisor-based security studies where they duplicate whole copy of the page tables. Secvisor [34] monitors the *cr3* register, an execution of *invlpg instruction*, and an occurrence of *page fault exception*, and finds when they have to synchronize page tables between the host and the guest.

**Physical Page Locking and Hiding:** Unauthorized access is not allowed on the physical address spaces that the hypervisor manages. Processes running in the guest's logical address spaces can make read-only access or none to protected physical pages. Figure 3 shows how addresses are translated by CPU and DMA. The address spaces of CPU, including the address space of the guest and the host, are translated to physical address spaces by memory management unit (MMU). The address space used by the guest machine, called guest logical address (GLA), is mapped to guest physical address (GPA) space via guest page table. CPU's memory virtualization is used to obtain final host physical addresses (HPAs) from GLAs. Light-box utilizes the page tables to redefine access permissions to the address spaces. The guest machine cannot reset the access permissions defined in hypervisor page tables (HPTs), and all accesses with wrong permissions are mitigated by Light-box. For example, if a guest process tries to write something on kernel codes, where only read and execute accesses are given in HPTs, Light-box intercepts the write operation and stops the guest.

Similar protection is applied to direct memory access (DMA). A DMA controller can access physical memory directly bypassing CPU's memory mitigation. Several studies have looked for the ways of dealing with DMA physical memory access attacks [31, 33, 36]. DMA accessible address spaces can be categorized into two: the DMA addresses recognized by devices and the addresses actually accessed by a DMA controller. DMA addresses are translated to HPA via input-output memory management unit (IOMMU) and DMA remapping table (DRT). DRT and HPT work in the same way.

Light-box protects physical memory by setting read-only permission or no permission in HPT and DRT. We call these techniques *physical page locking* and *physical page hiding*.

### 4.1.3 Privileged Register Protection

Modern operating systems differentiate modes of running according to required reliability, safety, or responsibility. For example, Linux and Windows operating systems have two modes of operation, in which tasks are running with different privileges. One is *kernel mode* (or supervisor mode), which corresponds to the *Ring 0* of the concept of the traditional protection rings. A wide range of system management tasks are done in this mode. Kernel mode tasks have unlimited accesses to the system, including kernel spaces. The other is *user mode*, which corresponds to the *Ring 3*. Casual applications are running in this mode. User mode applications are not allowed to modify kernel data.

There are special registers that require protections for the security of Shadow-box. Some of them are used to set access privileges on memory spaces. Some of them concern transitions between the kernel mode and the user mode. We call those registers, *privileged registers*, in the sense that they are only accessible through privileged instructions. The privileged registers are described as follows.

**GDTR and LDTR Protection:** Global descriptor table (GDT) is a set of segment descriptors and system descriptors. Each descriptor holds properties like base address, type, and limits. *Segment descriptors* are specified if the segment contains code, data, and stack. The access privilege to the segment is also specified in the descriptor. GDT has code and data segment descriptors for the kernel mode and the user mode. *System descriptors* include local descriptor table (LDT) descriptors, task state segment (TSS) descriptors, and call gate descriptors. LDT holds descriptors, similarly to GDT. While GDT can keep all sorts of segment and system descriptors, LDT only keeps segment descriptors and call gate descriptors. TSS stores information about task management, including processor's register state, I/O port permissions, stack pointers, etc. The call gate descriptor, or callgate, stores information to invoke codes across the privilege modes, such as address, number of arguments, and types.

The *GDT register (GDTR)* and the *LDT register (LDTR)* point to GDT and LDT, respectively. The values of GDTR, LDTR have to be handled properly via controlled and carefully designed procedures, and should not be altered with malicious intention. Light-box investigates the values of those registers and confirms if those values have not altered in an unauthorized manner, in the event-driven way by using CPU VT. Whereas GDTR and LDTR store values that are immutable, GDT and LDT store mutable values that are updated whenever task switching occurs. Light-box periodically traverses the descriptors stored in GDT and LDT, and tests the properties like the type and address range of each descriptor. By doing so, Light-box would recognize if tables are altered unexpectedly by malware.

**IDTR Protection:** *The IDT register (IDTR)* stores the address and size of interrupt descriptor table (IDT) that has vectors to the handlers, called interrupt gates and trap gates that handle interrupts and exceptions, respectively. Interrupts and exceptions are handled in the kernel mode, and affect the state of security. For example, handling `int 0x80` system call accords with privilege escalation. IDTR is protected by the event-driven way, similarly to GDTR and LDTR. Differently from GDT and LDT, the value of IDT is fixed once after it is set in the booting process. Light-box prohibits IDT from being altered, by setting up the memory area read-only.

**MSR Protection:** System calls are interfaces enabling user-level applications to access system resources. Traditionally, operating systems have provided a way of system call invocation via interrupts, although context switching overheads arise while handling interrupts.

Recent CPUs provide a better way of implementing system call interfaces being equipped with new instruction sets of `SYSENTER/SYSEXIT` and `SYSCALL/SYSRET`. The following *Mode Specific Registers (MSRs)* are needed to be set for using the new instructions sets. The SYSENTER_CS, SYSENTER_ESP, and SYSENTER_EIP registers are used to initiate kernel-mode execution and set up entry points of `SYSENTER/SYSEXIT` instructions. The STAR, LSTAR, and FMASK registers need to be prepared for `SYSCALL/SYSRET` instructions. By monitoring the values of MSRs, Light-box eliminates unauthorized mode transitions.

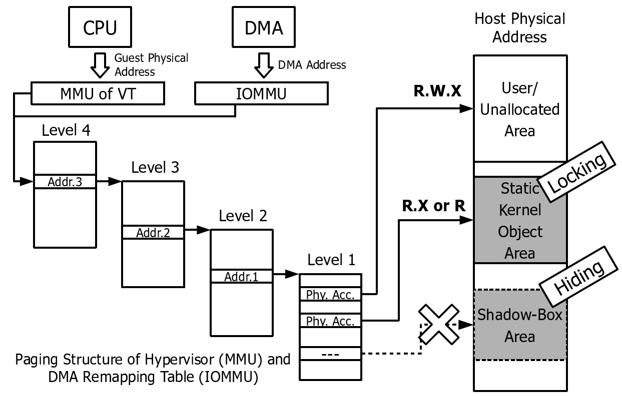


**Figure 4.** Event-driven access mitigation mechanism

#### 4.1.4 OS Independent Execution Flow

To implement monitoring procedures, we need to spawn control flows that are independent to the guest OS. Kernel threads could be used to create such control flows, but other kernel-level processes of the guest OS may intervene the threads. Instead, Light-box spawns OS independent control flows using the VMX preemption timer supported by CPU [11]. The VMX preemption timer can activate our monitoring logic periodically and give the control back to CPU afterhand. It is free from the guest's intervention, being running in the host machine.

### 4.2 Shadow-Watcher Design

For retaining control on the system permanently, malicious codes, such as rootkits, try to modify the critical kernel objects enlisted in Section 2. Protections on those objects are performed in an event-driven way and also a periodic way.

#### 4.2.1 Event-driven Access Mitigation

Kernel objects including kernel codes, the system call table, the IDT table, and the hypercall table, reside in read-only kernel memory. The values of the objects are static, thus immutable at runtime. The codes and read-only data of LKMs also fall into the same category. Shadow-watcher protects those objects by using *physical page locking*. As well as the locking, Shadow-watcher also uses *physical page hiding* for keeping the important objects safe.

When CPU or a DMA controller tries to access particular addresses, MMU and IOMMU translate given logical addresses to host physical addresses (HPA) using page tables shown in Figure 4. HPA may belong to a memory area allocated for static kernel objects, or a memory area used by the user or Shadow-box. Unintentionally or intentionally, HPA could also point to an unallocated memory area. Shadow-watcher sorts out those anomalies in memory accesses by re-setting proper access privileges in the pages tables.

- The static kernel objects (listed in Section 2) correspond to codes and data. Shadow-watcher sets read (R) and

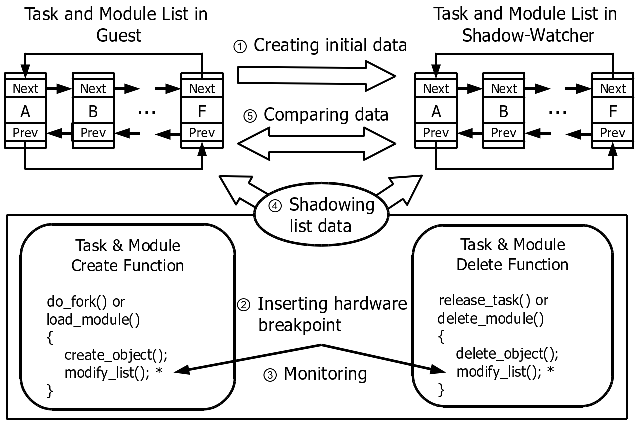


**Figure 5.** Operation of list shadowing

execution (X) rights for where the codes in, and only read (R) for read-only data.

- Shadow-watcher does not provide mapping to where Shadow-box codes and data are.
- No access limit is on user-area and unallocated area. All of read (R), write (W), execution (X) accesses are allowed.

### 4.2.2 Periodic Security Monitor

The dynamic kernel objects listed in Section 2 store mutable values frequently updated in runtime, therefore it is not practical to make them to read-only.

**List Shadowing:** Rootkits tend to modify system management data, such as the *task list* and the *module list*. They can hide out by deleting themselves in the double-linked lists, while running. Since rootkits turn themselves into the stealth mode on-demand, it is not predictable at all when they modify the relevant data. For detecting rootkits, Shadow-watcher makes copies of the lists, and compares periodically the current status of the lists and the stored copy. The copies of the lists are made utilizing H/W breakpoints. H/W breakpoints can be set on any location in code and data area, and do not require kernel code modifications.

Figure 5 shows how Shadow-watcher shadow important lists. When Shadow-box is loaded, it duplicates the current task list and the module list. H/W breakpoints are set on the operations that manipulate those lists, for example, creating/terminating a task, loading/unloading a module. If a change occurs in the lists, the exception (`0x01 #DB`) is raised by a H/W breakpoint, and the Shadow-watcher reflects the change into the copy of the list. Having an OS independent running cycle (See Section 4.1), the Shadow-watcher finds inconsistencies between the current lists and their back-ups. It allows detecting when malware tries to modify the system resources.

**Function Pointer Validation:** Each of virtual file system (VFS) objects and socket objects holds a series of function pointers which defines the possible operations on the object. Calls to those functions can be hooked by malware and be redirected to the codes that an attacker implanted. Those hooks are used to forge and intercept invocation parameters and return values. Shadow-watcher periodically validates the integrity of those function pointer.

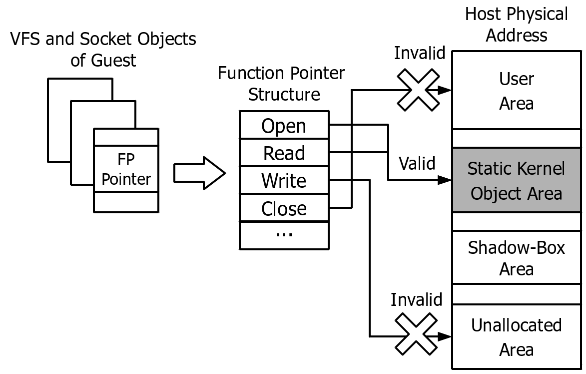


**Figure 6.** Operation of function pointer validation

As shown in Figure 6, VFS objects and socket objects store possible methods on the objects in a data structure, called function pointer structure. Each of the operation structure entry stores the entry point of handlers. The validity of those function pointers can be guaranteed if their addresses fall into the static kernel objects. Assuming Shadow-box is loaded securely to the memory, the entry points of the handlers should be inside the static kernel object (kernel code). Having entry points that point out unallocated area or user area, we can conclude that a malware came into the system and fabricated the function pointers.

## 5. Implementation

We explain how we implemented the Light-box and the Shadow-watcher, which are the key parts of the Shadow-box. It is implemented on an Intel machine for this paper, but also is feasible on any hardware that has the similar virtualization supports (e.g., recent AMD chips).

### 5.1 Light-Box Implementation

Light-box isolates the guest from the host, employing *on-access shadowing*, *physical page locking and hiding*, *privileged register protection*, and *OS independent execution flow*. How we implement those techniques are explained below.

**On-access Shadowing:** The shadow page table structure separating the host and the guest can be structured by `init_level4_pgt`. The `init_level4_pgt` stores the address of the top-most page table structure for the *init* process, called *swapper*. The *init* process stores only the mapping information of the kernel, so it is a good place to construct the shadow page table of Shadow-box. The constructed page table is stored in Host CR3 field of VMCS, so that the host will have a separated address space from the guest.

After the address spaces are separated, changes in the guest's page table would not be automatically reflected to the host's table. For the host to access the guest memory, we require mappings between the host's shadow page table and the GPA. The guest's top-most page table is stored in Guest CR3 fields of VMCS. This can be utilized by the host to look up GPA and update it to the host's shadow page table.

**Physical Page Locking and Hiding:** Physical page protection technique leverages the extended page table (EPT) of CPU VT and the second level page table (SLPT) of I/O VT. EPT is used to convert GPA to HPA, and its address is stored in the EPT Pointer field of VMCS. We activate EPT setting the Enable EPT bit in the Secondary Processor-Based VM-execution Controls field in VMCS. We set EPT could map the whole addresses to the guest, because the host and guest share the memory space. Similarly, we create SLPT to map the whole space of RAM. We activate SLPT using DMA remapping reporting (DMAR) tables in advanced configuration and power interface (ACPI).

After we create EPT and SLPT, we can set access privileges to 4KB units of physical pages and decide whether the pages can be mapped. We implemented *physical page locking* by giving read-only access privileges which depends on the characteristics of the pages, as mentioned in Section 4.1. We implemented physical page hiding technique by giving no access privilege on the pages.

**Privileged Register Protection:** CPU VT passes events to the hypervisor on VM exits, when the guest tries to access the *privileged registers* which include GDTR, LDTR, IDTR, and MSRs. The hypervisor can examine every access on *GDTR, LDTR, and IDTR*, receiving the access events by setting up the Secondary Processor-Based VM-execution Controls field in VMCS. We can also receive events when values of MSRs are changed. What we need to do is turning on the Use MSR bitmaps bit (bit 28) of Primary Processor-based VM-execution Controls field, and setting IA32_SYSENTER_CS MSR (0x174), IA32_SYSENTER_ESP MSR (0x176), IA32_SYSENTER_EIP MSR (0x175), IA32_STAR MSR (0xC0000081), IA32_LSTAR (0xC0000082), IA32_FMASK (0xC0000084) to 1 in MSR bitmaps. Table 2 summarizes how Light-box protects privileged registers and tables.

**OS Independent Execution Flow:** The VMX-preemption timer passes the control to the hypervisor periodically. VMX-preemption timer is activated by setting up the ticks on the VMX-preemption timer value field of VMCS, and by turning on Activate VMX-preemption timer bit (bit 6) of the Pin-Based VM-execution Control field in VMCS.

| Name | Protection method |
|---|---|
| GDTR/LDTR<br>IDTR/MSRs | event-driven mitigation<br>(using the VM-execution control) |
| GDT/LDT | periodic monitor<br>(verifying descriptors in the table) |
| IDT | event-driven mitigation<br>(locking physical pages) |

**Table 2.** Methods of privileged register protection

### 5.2 Shadow-Watcher Implementation

Shadow-watcher provides protections on the static kernel objects and the dynamic kernel objects, employing *event-driven access mitigation* and *periodic security monitor*. The implementation of those techniques explained below.

**Event-driven Access Mitigation:** Figure 7 shows the static kernel objects that we concern. The static kernel objects in the kernel space include the kernel codes, exception tables, and read-only data, as depicted in the shaded area. Those memory areas can be calculated using kernel symbols; the code area can be calculated by _text/_etext, the exception tables can be located by __start___ex_table/__stop___ex_table, and read-only data can be found by __start_rodata/__end_rodata. The kall_syms_lookup_name() function in the kernel returns the address of the symbols. The *physical page locking* technique defeats unauthorized accesses to those areas.

In case of loadable kernel modules (LKMs), a protection needs to be provided for the codes and read-only data of modules and their initializers, as shown in Figure 7. LKMs are connected in the form of linked-list and the modules symbol points to the head of the list. Each module of the list has the base address fields that point to start address and the size fields of the area. For example, the module_init field and the module_core field have the base address of area, and the init_ro_size field and core_ro_size field have the size of the read-only area. Note that the initializer codes are freed right after a module is initialized, so there is no need of protection for those codes. We only protect module_core area by using *physical page locking*.

**Periodic Security Monitor:** The H/W breakpoints for shadowing the task list and the module list are installed on exported functions that manipulate those lists, as follows: do_fork() and release_task() functions add and delete tasks. In do_fork(), a H/W breakpoint is set on wake_up_new_task() function which is invoked in do_fork() function. In release_task(), a breakpoint is set on proc_flush_task() which remove the task from /proc directory. Similarly, H/W breakpoints were set on the functions that add and delete a module to the module list. In load_module(), a breakpoint is set on ftrace_module_init() which is invoked after a module is added to the

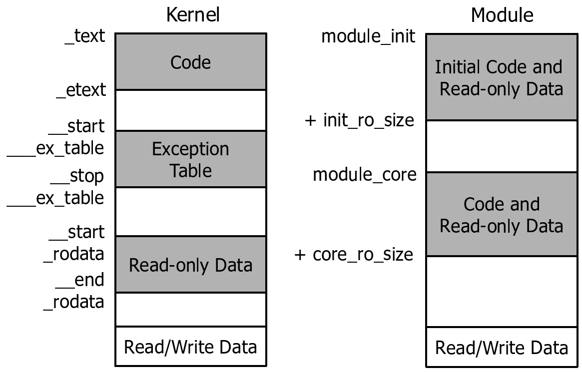


**Figure 7.** Static kernel objects in kernel area

list. In `free_module()`, a breakpoint is set on the start of `free_module()`.

We can trace changes in the task list and the module list using the H/W break points. We update the copies of those lists stored in Shadow-watcher, and then compare the actual lists and the copies periodically using the *OS independent execution flow*. The starting points of the task list and the module list are defined in *init_task* and *module* symbols, respectively. Those lists are double-linked list, easily retrieved by following the next pointer.

Function pointers are defined in VFS objects and socket objects. Function pointers are defined as file_operations structure and inode_operations structure, and stored in the `f_ops` field and the `i_ops` field of VFS objects. Many rootkits try to modify such VFS objects as the root directory and the proc directory. We periodically test if `f_ops` and `i_ops` field of those VFS objects point to the static kernel objects. Socket objects have the `d_inode` field and the `ops` field where function pointers are stored. The `d_inode` field is defined in tcp_seq_afinfo structure or udp_seq_afinfo structure. The `ops` field is defined in proto_ops structure. We periodically test if `d_inode` field and `ops` field of socket objects point to the static kernel objects.

# 6. Evaluation

We tested how well Shadow-box detects rootkits, and we also checked its performance. The performance evaluation was done on a desktop computer equipped with Intel i7-4790 3.6GHz, 32GB RAM, and 512GB SSD.

## 6.1 Rootkit Detection

We installed 64bit Ubuntu Hardy Heron with the Linux 2.6.24 kernel to run all five rootkits that we have. After running the rootkits, Shadow-box successfully detected all of them when they alter a bit of the kernel, as shown in Table 3 and Table 4.

As previously described, Shadow-box protects the static kernel objects in the event-driven way, while the dynamic kernel objects are validated periodically. It entails that alteration on the static objects are detected before dynamic ones are forged. Consequently, when Shadow-box runs with the proposed static and the dynamic kernel object protection, the rootkits are likely detected by the static kernel protection, as shown in Table 3. In order to show the effectiveness of the periodic security monitor, we intentionally turned the static kernel object protection off. Table 4 shows that Shadow-box still detects the rootkits. *Suckit 2.0* was exempted from the test because it does not modify any dynamic kernel object.

| Name | Detected? | Detected point |
|---|---|---|
| EnyeLKM | ✓ | code change |
| Adore-ng 0.56 | ✓ | function pointer change |
| Sebek 2.0 | ✓ | system table change |
| Suckit 2.0 | ✓ | system table change |
| kbeast | ✓ | system table change |

**Table 3.** Rootkit detection results-static and dynamic kernel object protection features are enabled

| Name | Detected? | Detected point |
|---|---|---|
| EnyeLKM | ✓ | module hide |
| Adore-ng 0.56 | ✓ | function pointer change, module hide |
| Sebek 2.0 | ✓ | module hide |
| kbeast | ✓ | module hide |

**Table 4.** Rootkit detection results-only dynamic kernel object protection feature is enabled

Specifically, Table 3 shows that *Adore-ng* is detected as it forged function pointers. In recent Linux kernels built-in drivers, function pointer structures are declared by `const`, and their instances are placed in read-only area. Attempts to change the instances are captured by the static kernel object protection. Besides, *Adore-ng* is also detected by the periodic monitoring (dynamic kernel object protection), because the pointers to the function instances are categorized into the dynamic kernel objects.

## 6.2 Performance Measurements

We evaluated performance of Shadow-box on Fedora 21 with Linux 3.17.4 kernel, separately on single-core processor and multi-core processor settings. After repeating benchmark five times, we calculated an average of the results. Our measurements were obtained using lmbench 3.0-a9, SPEC CPU 2006, and PARSEC 3.0 [16]. Outcomes of application benchmark tests are depicted in Figure 8.

**Single-core processor benchmark:** Table 5 and Table 6 shows the micro benchmark results conducted by lmbench in the single-core processor setting.

Overheads came from protection mechanisms of Shadow-box, as shown in Table 5. The mechanisms cause increased latency in 22.2-38.2%. The latency causes overall perfor-

| Name | Bare-metal | Shadow-box | Overhead |
|---|---|---|---|
| 2p/0K | 0.614 | 0.792 | 29.0% |
| 2p/16K | 0.564 | 0.708 | 25.5% |
| 2p/64K | 0.516 | 0.686 | 32.9% |
| 8p/16K | 0.728 | 0.900 | 23.6% |
| 8p/64K | 0.956 | 1.280 | 33.9% |
| 16p/16K | 0.872 | 1.066 | 22.2% |
| 16p/64K | 0.938 | 1.296 | 38.2% |

**Table 5.** Lmbench benchmark result-context switching latency (microseconds)

| Name | Bare-metal | Shadow-box | Ratio |
|---|---|---|---|
| Pipe | 8416.8 | 7832.8 | 93.1% |
| AF UNIX | 9167.6 | 8730.2 | 95.2% |
| TCP (Local) | 5952.6 | 4092.4 | 68.7% |
| File reread | 9593.58 | 9511.5 | 99.1% |
| Mmap reread | 15.72K | 15.2K | 96.7% |
| Bcopy (libc) | 10.6K | 10.46K | 98.7% |
| Bcopy (hand) | 6954.02 | 6868.7 | 98.8% |
| Mem read | 15K | 14K | 93.3% |
| Mem write | 10.26K | 10.14K | 98.8% |

**Table 6.** Lmbench benchmark result-communication bandwidth (MB/s)

mance overheads, and decreases the communication bandwidth. As shown in Table 6, other than TCP (Local) Shadow-box guaranteed at least 93% of network performance of the bare-metal machine, albeit TCP (Local) bandwidth went down about 32%. In order to find the reason, we placed H/W breakpoints on bare-metal machine, and measured the TCP bandwidth. Table 7 shows that how many overheads come from H/W breakpoints. The first row of the table shows about 6GB information could be transferred locally on the bare-metal machine without setting any H/W breakpoint. The second row presents the decrease of the bandwidth on the same machine with setting four of H/W breakpoints on. Comparing the results shown in Table 6 and Table 7, we can see H/W breakpoints decrease the bandwidth on bare metal as well. It seems H/W breakpoints consumes CPU cycles, and resulting in TCP bandwidth decreased. However, it was not significant in application benchmarks.

| Host | Bandwidth | Normalized |
|---|---|---|
| Bare-metal | 5952.6 | 1.00 |
| Bare-metal with H/W BP | 4197.8 | 0.71 |

**Table 7.** Comparisons of TCP (Local) bandwidth (MB/s)

Application benchmark tests were done using SPEC CPU 2006 and PARSEC 3.0, and we also measured compilation time of a Linux kernel. As shown in Table 8, Shadow-box introduces overheads of 6.4% on average (except PARSEC), which seems to be acceptable in most cases.

| SPEC CPU 2006 results | | |
|---|---|---|
| Host | INT (Sec.) | FP (Sec.) |
| Bare-metal | 3129 | 4133 |
| Shadow-box | 3439 | 4304 |
| Overhead | 9.9% | 4.1% |
| Kernel compile results | | |
| Host | Time (Sec.) | |
| Bare-metal | 2391.0 | |
| Shadow-box | 2517.8 | |
| Overhead | 5.3% | |
| PARSEC results | | |
| Host | Time (Sec.) | |
| Bare-metal | 1750.6 | |
| Shadow-box | 1773.9 | |
| Overhead | 1.3% | |

**Table 8.** Application benchmark results-single-core processor

**Multi-core processor benchmark:** Application benchmark was done on multi-core processor setting. Kernel compile time and PARSEC were measured, as shown in Table 9. On the multi-core processor setting, application benchmark reports 6.2% of overheads (except PARSEC).

| Kernel compile results | |
|---|---|
| Host | Time (Sec.) |
| Bare-metal | 4396.5 |
| Shadow-box | 4670.7 |
| Overhead | 6.2% |
| PARSEC results | |
| Host | Time (Sec.) |
| Bare-metal | 3071.5 |
| Shadow-box | 3097.1 |
| Overhead | 0.8% |

**Table 9.** Application benchmark results-multi-core processor

**VMX-preemption timer overhead:** Figure 9 depicts the changes in communication bandwidth along the frequency of VMX preemption timer. As it shows, frequent monitoring more than 10ms affects the performance of the system. In other words, we can run periodic monitor every 10ms without having significant performance decrease, but the optimal frequency might vary along the settings.

## 7. Discussion

Shadow-box does not guarantee control-flow integrity, which means that we do not provide an immediate countermeasure

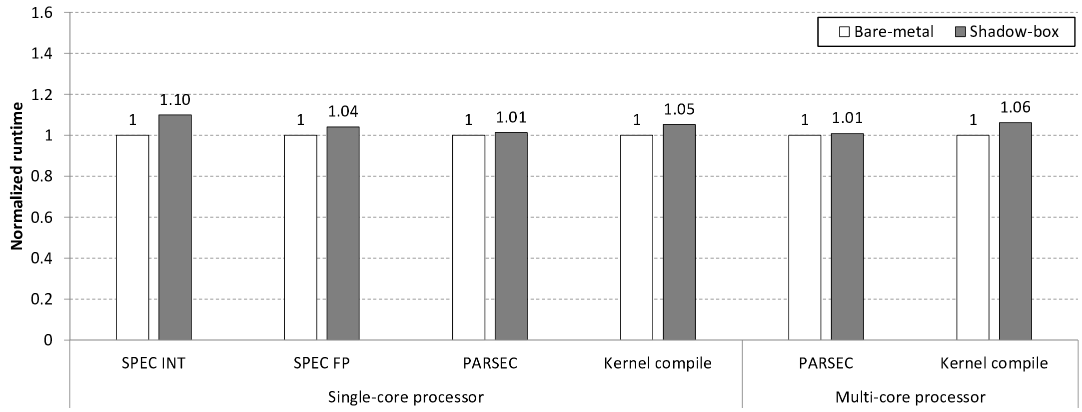


**Figure 8.** Results of application benchmark. Lower is better.

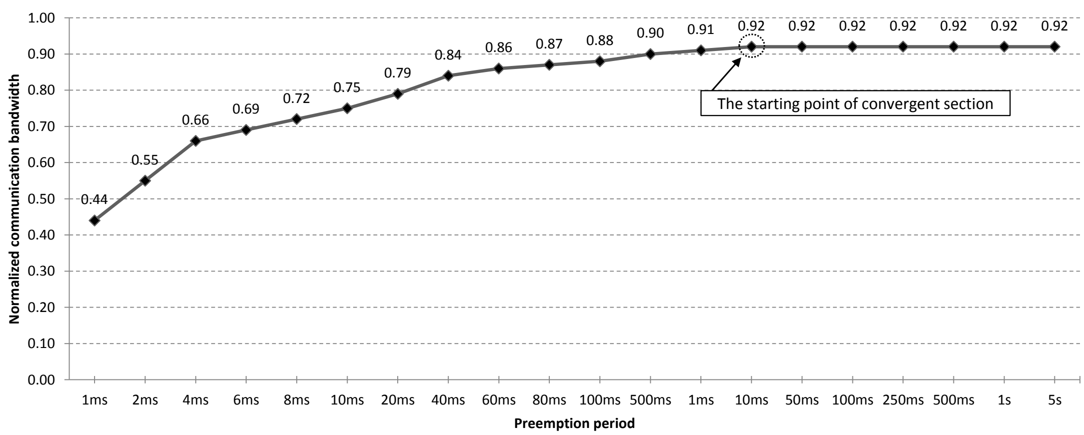


**Figure 9.** Changes in communication bandwidth according to the frequency of VMX-preemption timer

to such recent attacks based on return-oriented programming [20, 35]. However, if the attacks are aiming at forging or stealing valuable information in a system, they end up with modifying critical kernel objects. Shadow-box defeats those modifications.

Shadow-box installs H/W breakpoints and it may increase performance overheads as shown in Section 6. The overheads can be reduced by replacing the breakpoints with S/W breakpoints or syscall table hooks. S/W breakpoints and syscall hooks, however, require kernel modification.

Shadow-box does not allow runtime changes in kernel. Necessary self-modified codes or runtime kernel patches should be applied before shadow-box is loaded, unless prohibited.

## 8. Related Work

There are several approaches of guaranteeing kernel integrity employing VT. SecVisor [34] guarantees kernel integrity based on a tiny-size hypervisor, monitoring user-mode to kernel-mode transitions and checking if the transitions are initiated from unauthorized entry points. NICKLE [30] detects kernel rootkits leveraging general purpose hypervisors such as KVM and virtual box. NICKLE keeps copies of kernel codes and the hash values of known LKMs, so that it can identify unauthorized kernel codes and modified modules.

Lares [25], OSck [19], HUKO [39] and NumChecker [37] also fall into the same category which uses a general purpose hypervisor. Lares leverages Xen to protect kernel and anti-virus software installed in the machine. OSck leverages KVM to give protections on control flows and all sorts of kernel data including the dynamic kernel objects. OSck can protect rootkits effectively. HUKO leverages Xen to protect the static and dynamic kernel objects, and also guarantees control flow integrity. Their subject-aware protection mechanism inspects control flows depending on whether it executes kernel codes, kernel modules, or user applications, with a few overheads. Numchecker leverages KVM and hardware performance counters (HPC) to detect kernel rootkits. Their timing-based mechanism can detect rootkits without kernel modification of guest OS, and has a few overheads.

Copilot [26], Vigilare [24], and KI-Mon [23] guarantee kernel integrity by employing additional hardware. They could impose less overheads and the hardware-based protection cannot be compromised, but increases cost and issue compatibility problems.

## 9. Conclusion and Future Work

A virtualization-based OS monitoring framework, Shadow-box, is presented in this paper. It guarantees security by investigating accesses to protected kernel objects and also validating the objects periodically. The security is enforced by a lightweight hypervisor, Light-box, therefore security-related functions would continue working even when the OS is compromised. Light-box is a sort of Type-2 hypervisor, which imposes lower overheads than other virtualization tools and can be applied to a system without OS re-installation. We demonstrated the use of Light-box by implementing a kernel integrity monitor, Shadow-watcher. It successfully neutralized all well-known rootkits that we have tested, with low overheads.

Shadow-box employs a lightweight hypervisor, it can be applicable to resource-limited computing devices like mobile terminals. Mobile terminals are typically connected to the internet and they are less powerful than desktop processors. Therefore Shadow-box for ARM processors needs additional features such as remote attestation mechanism and workload-concerned monitoring mechanism. We are going to port the Shadow-box on ARM architecture and provide protections for mobile terminals in further study.

## Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0236-15-1006, Open Source Software Promotion)

## References

[1] VMware ESXi. `https://www.vmware.com/products/esxi-and-esx/overview`.

[2] Microsoft Hyper-V. `https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx`.

[3] *McAfee Labs Threats Report May 2015*. McAfee, .

[4] *McAfee Labs Threats Report March 2016*. McAfee, .

[5] ARM TrustZone. `http://www.arm.com/products/processors/technologies/trustzone`.

[6] Oracle VirtualBox. `http://www.virtualbox.org`.

[7] VMware Workstation. `http://www.vmware.com/products/workstation`.

[8] *Intel Virtualization Technology for Directed I/O*. Intel, 2014.

[9] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices, 2015.

[10] *AMD I/O Virtualization Technology (IOMMU) Specification*. Advanced Micro Devices, 2015.

[11] *Intel 64 and IA-32 Architectures Developer's Manual: Combined Vols. 1, 2, and 3*. Intel, 2015.

[12] *Intel Trusted Execution Technology*. Intel, 2015. `https://software.intel.com/en-us/articles/intel-trusted-execution-technology`.

[13] *Unified Extensible Firmware Interface Specification*. Unified EFI, Inc, 2016.

[14] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proc. of Computer Security Applications Conference*, pages 77–86. IEEE, 2008.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37 (5):164–177, 2003.

[16] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[17] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2009.

[18] J. Hartman. Verified Boot. 2009. `https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot`.

[19] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. *ACM SIGPLAN Notices*, 46(3):279–290, 2011.

[20] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc. of USENIX Security Symposium*, pages 383–398. USENIX Association, 2009.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proc. of the Linux Symposium*, volume 1, pages 225–230. Linux Foundation, 2007.

[22] P. Kleissner. Stoned bootkit. *Black Hat USA*, 2009.

[23] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proc. of USENIX Security Symposium*, pages 511–526. USENIX Association, 2013.

[24] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proc. of ACM Conf. on Computer and communications security*, pages 28–37. ACM, 2012.

[25] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proc. of Security and Privacy*, pages 233–247. IEEE, 2008.

[26] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proc. of USENIX Security Symposium*, volume 13, pages 179–194. USENIX Association, 2004.

[27] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[28] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *Proc. of ACM symposium on Applied computing*, pages 276–283. ACM, 2007.

[29] D. X. R Riley, X Jiang. Multi-aspect profiling of kernel rootkit behavior. In *Proc. of ACM European Conf. on Computer system*, pages 47–60. ACM, 2009.

[30] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proc. of Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2008.

[31] J. Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. *Black Hat DC*, 2007.

[32] J. Rutkowska and A. Tereshkin. Bluepilling the xen hypervisor. *Black Hat USA*, 2008.

[33] J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions. *Black Hat USA*, 2008.

[34] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[35] sqrkkyu and twzi. Attacking the Core : Kernel Exploiting Notes. Phrack #64 file 6. `http://phrack.org/issues/64/6.html`.

[36] P. Stewin and I. Bystrov. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2013.

[37] X. Wang and G. Xiaofei. Numchecker: A system approach for kernel rootkit detection and identification. *Black Hat Asia*, 2016.

[38] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proc. of Recent Advances in Intrusion Detection*. Springer, 2008.

[39] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proc. of Network and Distributed System Security Symposium*, 2011.

[40] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Proc. of Network and Distributed System Security Symposium*, 2008.

[41] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proc. of ACM SIGOPS European workshop*, pages 239–242. ACM, 2002.