Virtual Wall: Filtering Rootkit Attacks To Protect Linux Kernel Functions

Yong-Gang Li, Yeh-Ching Chung, Kai Hwang, *Life Fellow, IEEE*, and Yue-Jin Li

Abstract—Linux servers are being used in almost all clouds, datacenters and supercomputers today. Linux Kernel functions are facing a kind of malware attacks, known as rootkits with root-access capability. The rootkits appear as *loadable kernel modules* (LKM) in today's Linux servers. These modules hide from other kernel objects, and can redirect the kernel control flow by tampering with the metadata needed in kernel service functions. The kernel rootkits are invisible to users after loading, which may bypass most security shields. Both spatial and temporal appearance of rootkits are randomly distributed, which makes it difficult to detect or removal. To deal with rootkit threats, we propose a novel *Virtual Wall* (VTW) approach to filtering out the rootkit-embedded LKMs by tracing the incurred kernel activities. This VTW is essentially a lightweight hypervisor built with rootkit detection and event tracing capabilities. Normally, the Linux runs in a guest mode. When a LKM execution violates the security policy set by the VTW, the OS control will switch to a host mode. The VTW at host mode enables the detection and tracing of rootkit events timely. In other words, potential rootkit attacks are detected, traced and classified to make meaningful filtering decisions. The whole detection and tracing process is based on memory access control and event injection mechanisms. Experimental results show that the VTW defense system is effective to detect and defend against kernel rootkits timely. The CPU overhead for executing VTW is less than 2 percent. Compared with other defense schemes (such as DIKernel, etc.), our vs is easier to implement with low performance degradation on Linux servers. We will demonstrate the advantages of VTW through its simplicity in implementation and potential performance gains. We will also compare our system with seven other rootkit defense systems.

Index Terms—Access controls, data integrity, OS security, kernel protection and system architectures

1 Introduction

Kernel rootkits [1], [2], [3] are widely used in kernel attacks in the Linux servers due to their high privilege and hidden features. Currently, the known kernel rootkits appear mostly in the form of *Loadable Kernel Modules* (LKM) [4]. These modules can redefine kernel component functions, hide themselves, and hide target objects [5].

The hidden feature allows kernel rootkits to bypass the security tools with "host-based" architecture [6], [7]. Since the lower privilege of these tools limits their detection scope. Worse, their dependence on the *target operating system* (TOS) environment leads to the fact that the authenticity of detection results must be based on TOS's absolute security. Unfortunately, kernel rootkits can sneak tampering with TOS environment to destroy its security.

To achieve their functionality, kernel rootkits need to tamper with kernel objects. The operating objects of kernel rootkits cover various data such as control data, non-control data, static data, and dynamic data [8]. Theoretically, the kernel security can be guaranteed by limiting all the tampering operations to certain kernel data.

However, different data are usually stored decentralized in kernel space, and data with different attributes may be stored in a same memory page. Therefore, the page-based memory protection mechanism cannot protect the target data without affecting other kernel data.

During attack, the kernel data that kernel rootkits need to tamper with is often only a few bytes. Meanwhile, the minimum granularity of memory privilege management is one page, which typically occupies 4 kb. That is, in order to protect a few bytes, we need to limit all executable entities' writing operations to the page where the target data locates.

The mismatch between the data size and the protection granularity destroys the original attributes of other kernel data, which affects the functionality and performance of the associated execution entity. For example, the VFS function pointer entry in kernel data structure typically is the target data to be tampered with by kernel rootkits [9]. Although this entry is not dynamically updated, the status description entry co-located with it may be updated as the execution entity running. If the write access to the status description is restricted, the functionality of the associated execution entity will be affected.

Kernel rootkits have all the LKM features. They can call the custom functions or the loaded rootkits' exported functions to tamper with kernel data. They can launch an attack during loading time or at any time within their lifecycle. Therefore, the kernel rootkit may be any LKM being loaded

- *Yong-Gang Li is with the School of Data Science, the Chinese University of Hong Kong, Shenzhen, China, and the University of Science and Technology of China, Anhui 230052, China, and also with the Shenzhen Institute of Artificial Intelligence and Robotics for Society, Guangdong 518172, China. E-mail: liyonggang@cuhk.edu.cn.*
- *Yeh-Ching Chung, Kai Hwang, and Yue-Jin Li are with the School of Data Science, the Chinese University of Hong Kong, Shenzhen, China, and also with the Shenzhen Institute of Artificial Intelligence and Robotics for Society, Guangdong 518172, China. E-mail: {ychung, hwangkai, 218019063}@cuhk.edu.cn.*

Manuscript received 17 June 2020; accepted 16 Aug. 2020. Date of publication 4 Sept. 2020; date of current version 8 Sept. 2021.
(Corresponding author: Kai Hwang.)
Recommended for acceptance by N. Abu-Ghazaleh.
Digital Object Identifier no. 10.1109/TC.2020.3022023

or already loaded, and the kernel rootkit(s) that participate in an attack may be one or multiple LKMs.

From the time perspective, the attack time of kernel rootkits is random. From the space perspective, the participants in an attack are scattered. The two characteristics makes the attack time of the kernel rootkits difficult to predict, and the attack participants difficult to trace, which affects the system security seriously.

The randomness of the attack initiation time makes the method based on single or periodic detection can only achieve the effect of post hoc detection. When these methods are executed, kernel rootkits' attack may have been completed, even all the rootkits may have been removed. As a result, their detection and defense effects will be seriously affected. The dependency between kernel rootkits makes the attack form more diversified.

Through the dependencies, kernel rootkits can concatenate multiple LKMs for kernel attack. The existing security methods can only detect the direct initiator of the current attack, but cannot identify other participants in the entire attack process. Undetected attackers will always be lurking in the OS, waiting for launching attacks again when conditions are ripe.

We propose a kernel rootkit filtering method VTW to protect the kernel functions such as the system call table, read only data of kernel, state descriptors of LKM, "*proc*" file function pointers, and network function pointers, etc. This method uses Intel VMX technology to divide the OS into two modes: "host" and "guest".

Across the two execution modes, we create detection, prevention, and tracing strategies for compacting kernel rootkits. The main contributions of this paper are summarized into four technical aspects:

1) Build a lightweight hypervisor. It uses Intel VMX to reconstruct the OS's execution mode, making the resource access of kernel more detectable.
2) Establish a resource access control mechanism. It can realize the perception, limitation and manipulation to the target memory access, and support us in monitoring and controlling the execution paths of "guest" mode.
3) Establish a control flow tracing mechanism. This mechanism support tracing the control flow's jump and return among different LKMs.
4) Create a secure framework to detect, defense and trace kernel rootkits. Through the above security architecture and mechanisms, the kernel rootkits attack can be detected and defensed. For the first time, a tracing method is developed to trace all the participating kernel rootkits in an attack.

2 Related Previous Work

The kernel protection methods based on virtualization have received extensive attention due to their good anti-interference [10], [11]. Especially for kernel rootkits, these methods have shown outstanding advantages. They can be divided into two types: the intrusive methods and the non-intrusive methods. The former needs to inject additional components into the *target virtual machine* (TVM) to get the required information. The latter does not need to do so.

Intrusive Methods. X-TIER [12] proposed by Sebastian injects a kernel module into a target virtual machine first. It then reads the data structures of TVM through the module to obtain its status information. After that X-TIER passes the acquired information to hypervisor through hypercall.

SYRINGE [13] uses function-call injection technology to enable calls to TVM's functions outside of TVM. At the same time, localized shepherding technology is used to monitor the integrity of the control flow.

Virtuoso [14] proceeds to obtain TVM status information from a control logic perspective. It runs the program multiple times in TVM, and extracts relevant instructions and instruction execution paths. Then, the path required to generate the introspection code is translated. Finally, Virtuoso translated all information into code which can perform semantic refactoring outside of TVM.

X-TIER, SYRINGE, and Virtuoso get the semantic views by analyzing physical memory and treat them as real views. They can find the hidden objects such as processes and files by comparing the real semantic views with TVM's internal view. If there is a hidden object, they judge the TVM has been compromised and a kernel rootkit may exist in TVM. VMST [15] takes the same method for rootkit detection.

Non-Intrusive Methods. The first step of VMwatcher [16] is to obtain the memory of TVM. It then uses the kernel data structure of TVM as templates to interpret the operational status represented by the memory.

Unlike VMwatcher, RTKDSM [17] is a real-time system. It can be divided into two parts: the introspective agent and the monitoring agent. The former is placed in a secure virtual machine and the latter is placed in the hypervisor [18]. RTKDSM can realize real-time monitoring to the data structure by cross comparison.

The above methods can be used to detect the objects hidden by kernel rootkits. However, these approaches rely on large virtualization platforms such as Xen and they need to overcome the semantic gap through virtual machine introspection [19], [20], [21], [22], [23]. As a result, they introduce a significant performance overhead to the OS.

For example, RTKDSM slows down the execution speed of some applications by 110 percent. SYRINGE delays up to 51 ms on system calls, which is a significant performance overhead for system calls. Except for RTKDSM, none of the above methods can monitor and detect the OS in real time.

3 Virtual Wall Architecture

VTW is a lightweight hypervisor with the defense function of kernel rootkits. In this section we introduce its overall architecture.

3.1 Assumptions and Notational Definition

We assume that the attacker can inject an LKM into the TOS. In practice, the attacker can elevate its privilege through application vulnerabilities and control TOS with a backdoor [24]. Then an LKM can be injected into the kernel. We also assume that the TOS and non-malicious execution entities do not illegally tamper with kernel data and kernel control flows. Moreover, the definition of symbols used in this paper is shown in Table 1.

3.2 Design Objectives

The VTW is designed with three technical requirements as specified below:

TABLE 1
Notation and Definition Used in Article

Notation	Definition
$A::=(a,b,c)$	Meta data A with 3 components
$A:a$	Metadata a in the tuple A.
$A{+}{+}d$	Metadata d is added to tuple A.
$a.x$	Take the x in the metadata a.
$m\rightarrow n$	Replace data n with data m.
$E\not\succ R$	Result R by executing expression E.
$I\Vdash C$	Conclusion C derived from I.
$S\mapsto i$	Take the entry i in the data structure S.
$\triangleright$	Lead to subsequent execution.
$a\bowtie L$	Metadata associated with L.
$a\propto V$	Metadata located in the area V.
$V_1\leftarrow V_2$	Data in set V2 written into set V1.

1) *Real-time Detection.* Kernel rootkits may launch an attack at the module loading stage, or at any time during the module's reside time in memory. If the detection is performed after the LKM has been loaded or when the LKM has been removed, the detection result may become inaccurate, which will affect the subsequent defense and traceability. Therefore, VTW needs to detect rootkits' malicious operations before module loading completion or during their lifetime to protect kernel data integrity [25].
2) *Effective Defense.* The kernel objects manipulated by rootkits include static and dynamic kernel data. For static kernel data, VTW needs to ensure they have never been tampered with. For dynamic kernel data, VTW needs to ensure they can be restored after being tampered with.
3) *Comprehensive Traceability.* The attack code may be in the memory space to which the rootkit belongs, or it may be in other module code space that the rootkit depends on. VTW needs to locate all LKMs related to the current malicious operation.

3.3 System Architecture of Virtual Wall

The process of VTW defense is shown in Fig. 1. The initialization is needed to set up the defense pattern. Then the access control scheme is applied. Finally the detection and tracing are performed jointly and cooperatively to make fast decisions on rootkit filtering.

The architecture of VTW is shown as Fig. 2. VTW consists of ModeHandler, MemHandler, ControlHandler, and SPE

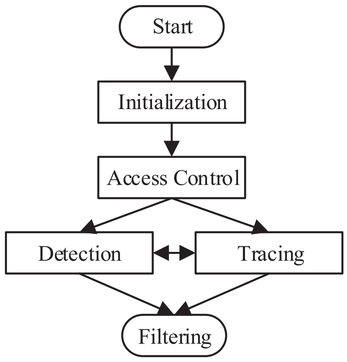


Fig. 1. The process of virtual wall operations.

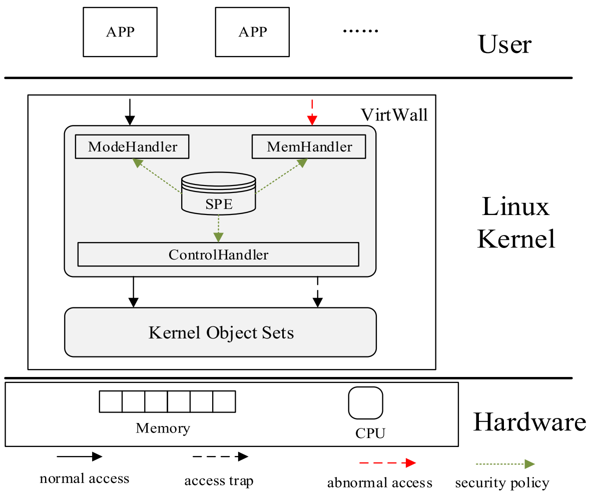


Fig. 2. VTW defense system architecture.

(Security Policy Engine). The task of ModeHandle is to ensure the normal mode switching between the "guest" and "host" modes.

The task of MemHandle is to isolate security components from the "guest" by establishing a set of separate addressing pagetables. In addition, it leverages EPT to implement VTW's self-protection and transparent deployment. ControlHandle performs rootkit detection, prevention, and analysis.

SPE provides security policies for each component, such as permission settings for static kernel data, legality constraints for dynamic kernel data, and tracing paths for control flow. All violations to SPE will cause the OS to fall into "host" from "guest".

3.3.1 Resetting of OS's Privilege Mode

ModeHandler uses Intel VMX Non-Root and VMX Root to divide the native OS into "guest" and "host" modes. These two modes divide ring0 into two privilege levels, which are called the full ring0 and the restricted ring0. In "host" mode, VTW is at full ring0 level, and it completely takes over the kernel control flow. In "guest" mode, the kernel is at the restricted ring0 level, and any operation that violates SPE will cause the OS to fall into "host" mode.

In Fig. 3, the ProtectionWall is an abstraction of the functional proper-ties for ModeHandler and MemHandler. KernelProtector is an abstraction of the ControlHandler's functional property.

If an instruction In guest mode with violation of the security policy, it will cause OS to switch to the host mode. The ProtectionWall will reject all operations in guest mode and wake up the KernelProtector to handle the attack. using the SPE.

3.3.2 Creating Private Page Tables

MemHandler creates a set of private pagetables for the host to separate the address space between the two modes. The conditions for the construction of the private pagetable are given in Table 2.

The kernel data *swapper_pg_dir* stored in kernel data segment points to the page directory of the kernel address

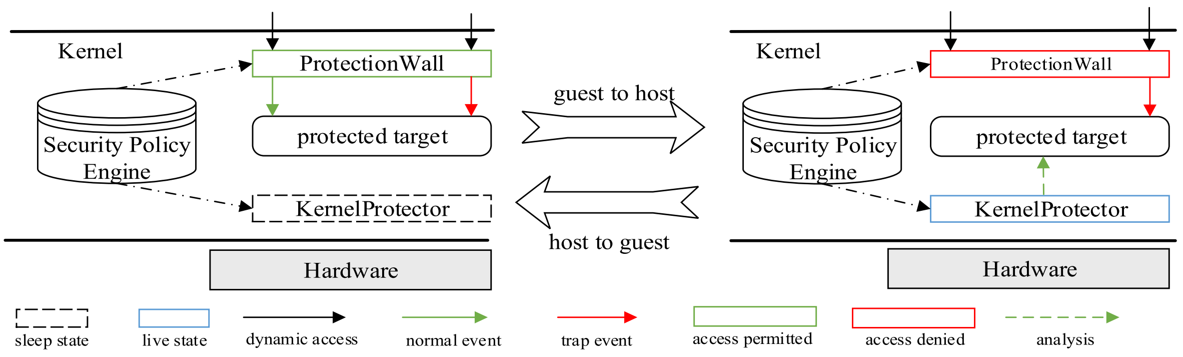


Fig. 3. Conditional switching between guest and host modes in Linux kernel operations.

space. The kernel address space of all execution entities is constructed based on *swapper_pg_dir*. In Table 2, all private pagetables form a tupleρ, and all pagetables pointed to by *swapper_pg_dir* form a tupleε (①∼②).

When MemHandler is initialized, it creates the page directories (*h_pml4e*), page upper directories (*h_pdpte*), page intermediate directories (*h_pde*), and pagetables (*h_pe*) according toε.

We first create an equal amount of memory V_1 based on the memory V_{22} of the page table set ρ (③). Then we fill *h_pml4e's* entry with the physical address of each *h_pdpte*, fill *h_pdpte's* entry with the physical address of each *h_pde*, and fill *h_pde's* entry with the physical address of each *h_pe* (④∼⑥).

Finally, MemHandler copies all the page entries in the last pagetables of ε to *h_pe* (⑦). When a pagetable error occurs in "host", VTW updates the error pagetable according toε.

To ensure VTW's transparency, MemHandler establishes an EPT redirection method. This method makes VTW's components are invisible to "guest". It redirects the EPT page entries corresponding to each component's physical memory of VTW to a blank page that is set to be readable, writable, and executable (⑧∼⑩). When the entity in "guest" intends to access VTW's physical memory, the accessed memory is a "*pseudo page*".

Since the entire kernel shares a same address space, each execution entity can map all kernel content, including VTW. Therefore, without any processing, VTW can be detected and analyzed. If we just delete the EPT entries addressing VTW, the OS will repeatedly fall into "host" when an entity probes the kernel address space.

TABLE 2
Required Conditions for Page Table Insulation

Operation	Condition setting
Private and public tables	① ρ:: = (h_pml4e, h_pdpte, h_pde, h_pe)
	② ε:: = (g_pml4e, g_pdpte, g_pde, g_pe)
	③ $\rho \propto V_1$ $\varepsilon \propto V_2$ $V_1 \leftarrow V_2$
Fill private tables	④ ρ:h_pml4e.pdpte_item→V_1[ρ:h_pdpte]
	⑤ ρ:h_pdpte. pde_item →V_1[ρ: h_pde]
	⑥ ρ:h_pde. pe_item →V_1[ρ: h_pe]
	⑦ ρ:h_pe. page_item →V_2[ε: g_pe.item]
Redirection permission setting	⑧ ρ:h_pe. host_item →pseudo_page
	⑨ pseudo_page.rw = true
	⑩ pseudo_page.x = true

On the one hand, frequent trapping will affect the execution efficiency; on the other hand, the abnormal time and space of memory access will cause the attacker to infer VTW's existence. Redirecting all address spaces of VTW to pseudo-physical pages can eliminate time and space anomalies caused by memory probing, which enhances the transparency of VTW.

4 Resource Access Control

VTW controls memory access through MemHandler. MemHandler uses EPT to control memory access in "guest" mode. Through the memory permission management provided by EPT, we can perceive, trace and control "guest" access to all memory pages. MemHandler is easy to deploy by setting the permission bits in last level of EPT.

To achieve more granular access control to kernel, ControlHandler establishes an event injection mechanism that includes setting breakpoints, injecting general protection exceptions, and opening single-step debugging. We will describe them in this section, which is shown as the Table 3 and "Resource Access Control Procedure".

4.1 Conditioning for Resource Access Control

The Table 3 is the conditions for setting resource access control. ①∼⑨ are used for setting breakpoints. The types of breakpoints can be divided into instruction and data breakpoints in terms of the target objects. ControlHandler uses Dr0∼Dr3 (breakpoint address registers in set *DeReg*) and Dr6∼Dr7 (breakpoint management registers, in the set *DeCon*) to set different types of breakpoints (①∼②).

When setting an instruction breakpoint, the instruction address (*Iaddr*) is first written into the breakpoint address register (③) through the function *SetDeReg*. Then the Dr7's Read/Write bit is set to 00, and the corresponding LEN bit is set to 00 (byte length is 1), which is shown as ④∼⑤.

When setting a data breakpoint, the address of the target data is first written to the breakpoint address register (⑥) through the function *SetDeCon*. Then, the corresponding R/W bit of Dr7 is set to 01 (data write interrupt), and the corresponding LEN bit is set to 10 (the data size is 8 bytes), which is shown as ⑦∼⑧. After completing the breakpoint setting, ControlHandler also needs to clear the B0∼B2 (bits 2:0) of Dr6 (⑨).

To control the execution of "guest", ControlHandler sets the single-step debugging for OS. ControlHandler first reads the contents of EFLAGS of the guest state field in

TABLE 3
Conditioning of Resource Access Control

Operations	Condition setting
Debug registers	① DeReg:: = (dr_0, dr_1, dr_2, dr_3)
	② DeCon:: = (dr_6, dr_7)
Instruction point setting	③ SetDeReg[DeReg:dr_m→Iaddr]
	④ SetDeCon[DeCon:dr_7:rw_0→bit:00]
	⑤ SetDeCon [DeCon:dr_7.len_0→bit:00]
Data point setting	⑥ SetDeReg[DeReg:dr_n→Daddr]
	⑦ SetDeCon[DeCon:dr_7.rw_1→bit:01]
	⑧ SetDeCon[DeCon:dr_7.len_1→bit:10]
dr6 setting	⑨ SetDeCon [DeCon:dr6.(b0∼b2)→0]
#GP setting	⑩ VM_interupt[bit(31)→bit:1]
	⑪ VM_interupt[bit(8∼10)→bit:011]
	⑫ VM_interupt[bit(0∼7)→bit: 00001011]
Single step setting	⑬ pre_status:: = (cpu_context, memory)
	⑭ EFLAGS.tf = Enabled
	⑮ dr6.bit(14) = Disabled

VMCS, records the target information (⑬). Then it sets EFLAGS.TF (the trap flag) to 1 (⑭), which puts the processor into single-step mode. Finally, ControlHandler writes the modified content to EFLAGS again. The BS (bit 14) bit of Dr6 is set to 0 at the end.

4.2 Resource Access Control Procedure

This procedure is used for controlling the instruction execution and data access. Steps 1∼4 are used to get the breakpoints; steps 5∼6 are used to block the target instruction in "guest"; steps 7∼8 are used to get the OS status after executing a target instruction.

```
1.  Read/Write(d) $\nsucc$ SysMod[guest→host]
2.  $\nsucc$ DeCon: dr6.(b0∼b2) = = m
3.  GuestExecute(i) $\nsucc$ SysMod[guest→host]
4.  $\nsucc$ DeCon: dr6.(b0∼b2) = = n
5.  GuestExecute(any_instruction) $\nsucc$ SysMod[guest→host]
6.  $\nsucc$ Current operation is blocked
7.  GuestExecute (any_instruction)$\nsucc$SysMod[guest→host]
8.  $\nsucc$get_status()
```

When an execution entity writes data into a breakpoint location (steps 1∼2) or executes a breakpoint instruction (steps 3∼4) in "guest" mode, it triggers the mode switching (*SysMod*) into "host" mode. The B0∼B2 in Dr6 are used to distinguish the position of the breakpoint. By setting breakpoints, VTW implement byte-level resource access.

When a general protection is set, OS will generate a #GP exception when running in "guest" mode, so that the current operation is blocked (steps 5∼6). When the single-step debugging opens, OS will trigger a debug exception and fall into "host" after executing any instruction in "guest" (steps 7∼8). By setting the single-step debugging, we can implement resource access control for OS with instruction granularity, and obtain the change of OS's state caused by each instruction through *get_status*.

5 Rootkit Defense Strategies

According to the attack time, we classify the attack of kernel rootkits into three categories. For the first type, rootkits attack kernel during their loading. They call custom functions via *module_init()* to realize their attack.

TABLE 4
Conditioning of Rootkit Attack Detection

Operation	Condition setting
Static kernel objects setting	① μ:: = (all_protected_static_pages)
	② ξ:: = (kernel_code, syscall_table, . . .)
	③ $\xi \subseteq \mu$
	④ ζ:: = (ept_tabes: protected_area_item)
	⑤ ζ.bit(1) = Disabled
LKM information	⑥ $\mathscr{M}$:: = (module_list)
	⑦ $\mathscr{M}$p = $\mathscr{M}$c $\mapsto$ prev
	⑧ $\mathscr{M}$n = $\mathscr{M}$c $\mapsto$ next
Dynamic kernel data setting	⑨K:: = (D1, D2 . . .. . . Di. . .. . .)
	⑩ Di:: = (ai, ci)⊆K
	⑪ X:: = (s, e)
	⑫ ci′ = ReadDataFromAddr(Di: ai)

The malicious behavior of kernel rootkits may occur in one or several of the above three types of attack. At the same time, the rootkit(s) associated with the current attack may be the LKM being loaded, either the loaded LKM, or both.

The VTW determines whether a LKM is a kernel rootkit. For kernel rootkits, VTW defends against them by blocking their corrupted actions and restoring the corrupted data. The rootkit detection method is shown as "Detection of Rootkit Attacks", whose condition setting is shown in Table 4. The method can be used to predetect the static and dynamic kernel objects.

Detection of Rootkit Attacks. It is used to detect kernel rootkits and protect the kernel from being destroyed. Steps 1-5 are used for static kernel object protection; steps 6∼8 are used for hiding detection; steps 9∼11 are used for dynamic kernel object protection.

```
1.  WriteTo($\xi$) $\nsucc$SysMod[guest→host]
2.  $\nsucc$InjectGP()
3.  WriteTo(C$\mu\xi$) $\nsucc$SysMod[guest→host]
4.  $\nsucc$SingleStep()
5.  $\nsucc$OpenWrite(TargetPage)
6.  $\mathscr{M}$p = = $\mathscr{M}$ n = = null $\vdash$ $\mathscr{M}$c is hidden
7.  $\mathscr{M}$p $\mapsto$ next $\neq \mathscr{M}$c $\vdash$ $\mathscr{M}$c is hidden
8.  $\mathscr{M}$n $\mapsto$ prev$\neq \mathscr{M}$c $\vdash$ $\mathscr{M}$c is hidden
9.  $\forall$ $a$i has X: $s \leq c$i′ $\leq$X: $e$ $\vdash$kernel is not destroyed
10. \$ $a$i has $c$i′> X: $e \| c$i′<X: $s$ $\vdash$kernel is destroyed
11. $\triangleright$Restore[Di. $c$i→$c$i′]
```

5.1 Static Kernel Object Protection

The static kernel objects ($\boldsymbol{\mu}$ and $\boldsymbol{\xi}$) remain unchanged in OS. The specific execution path of kernel control flow and some important data are stored in static kernel objects. Kernel rootkits can achieve their malicious purpose by breaking the integrity of static kernel objects [26].

To protect the static kernel objects (such as the OS code segment, data segment, system call table, etc.), VTW get their linear addresses by analyzing the file "*System.map*" or extracting from kernel data segment. All of them will be translated into physical addresses.

After that, VTW sets the write permission of the EPT entries ($\boldsymbol{\zeta}$) that index these physical addresses to "unwritable" (④∼⑤ in Table 4). In "guest", the write operation to (*WriteTo*) static kernel data will trigger "EPT violation" causing OS to fall into "host" (step 1 in "Detection of Rootkit Attacks").

Then, VTW injects a #GP exception (*InjectGP*) into "guest" to prevent LKM destroying kernel data (step 2).

It should be noted that the minimum granularity of the memory protection is one "page". However, not all static kernel objects occupy several complete pages. Therefore, when an "EPT Violation" exception is thrown, VTW first determines whether the exception address belongs to the address range of the protected objects.

If yes, the current operation will be blocked. If not, VTW will set "guest" to single-step mode (*SingleStep* in step 4). Then it sets the page as "writable" (step 5). Finally, VTW switches OS back to "guest" mode to complete subsequent writes. After that, OS re-traps "host", and VTW restores the page to "unwritable" through the function *OpenWrite*.

5.2 Dynamic Kernel Object Protection

Kernel rootkits can achieve malicious purposes such as hijacking the control flow by tampering with dynamic kernel data. The tampered kernel data includes both control data and non-control data. The former refers to reference pointers pointing to kernel control flow.

These pointers are usually used to construct semantic views, such as function pointers stored in "*proc*" file system. The latter refers to some entries in certain status descriptors, such as entry "*prev*" and "*next*" in "*struct module*".

The kernel data updated in real time and the target data tampered by rootkits may be stored in a same page. So we cannot prevent kernel rootkits from tampering with kernel data by limiting writing permission to the memory page.

In addition, the randomness of attack time makes it difficult to detect the tampering operation in real time. Faced with these problems, VTW traces the execution of LKMs, and detects the dynamic kernel data during LKM tracing.

5.2.1 Hidden LKM Detection

Self-hiding is the basic feature of a kernel rootkit. The hidden behavior can be treated as a criteria for judging whether the LKM is a rootkit. Typically, kernel rootkits implement self-hiding during their initialization.

Therefore, VTW needs to detect whether it has been hidden before initialization is completed. To trace module initialization we set an instruction breakpoint at *sys_init_module ()* and monitor LKM's state change. When the state is switched to *MODULE_STATE_LIVE*, VTW checks whether the module is hidden. Kernel rootkits break the logical connection with others by removing their status descriptor from a linked list for hiding purpose. The logical connection between LKMs can be used to determine whether the module is hidden.

$\mathscr{M}$c is the LKM being loaded. $\mathscr{M}$p is the module pointed by the entry *prev* in $\mathscr{M}$c, and $\mathscr{M}$n is the module pointed by the entry *next* in $\mathscr{M}$c (⑦∼⑧ in Table 4). If the module $\mathscr{M}$c does not have a linked list connection relationship with its neighboring nodes (step 6 in "Detection of Rootkit Attacks"), or the connection relationship is incomplete (steps 7∼8), it is judged that the module $\mathscr{M}$c is hidden.

5.2.2 Dynamic Kernel Data Detection

In addition to status descriptors, kernel rootkits also tamper with the kernel data with control properties for kernel control flow redirection. This type of kernel data consists of pointers that point to the kernel code segment before redirection. Kernel rootkits rewrite the pointers to redirect them to their custom code.

The redirected objects mainly include various operation functions for building OS semantic views. Function pointers (such as *lookup*) in "*proc*" file system are the objects most vulnerable to kernel rootkits. So it is necessary to protect these function pointers from being tampered with.

To hide network information (such as network ports), kernel rootkits also attack the data in "*/proc/net/tcp*", "*/proc/net/tcp6*", "*/proc/net/udp*", and "*/proc/net/udp6*". So the function pointers in these file descriptors are also need to be protected.

In addition to the above kernel objects, VTW also treats the function pointers of key files such as "*root*" and "*log*" files as protection objects. All the function pointers and their storage addresses can be extracted from memory according specific data structure (such as *f_dentry->d_inode->i_op->lookup*) during VTW initiation.

All the extracted data forms a set κ. All the dynamic kernel data in κ points to fixed kernel functions and they need not to be updated. Only when we find that a new kernel rootkit modifies a certain kernel data that is not included in κ, we expand κ with the certain data. Moreover, VTW will set κ as read-write protection through EPT, and prohibit the execution entity in "guest" mode from accessing κ to protect it.

Now, κ occupies about 64KB of memory and contains 4000 pieces of kernel data, and it will grow with the emergence of new rootkits. In practice, the existing kernel rootkits often tamper with no more than 500 pieces of kernel data. We expanded the scope for better protection.

Each element D in set κ corresponds to a unique piece of kernel data (⑨ in Table 4). The element Di exists as a binary data pair (ai, ci). ai represents the storage address of the protected data, and ci represents the data content (⑩). The X is the range of kernel code range $s\sim\ _e$ (⑪).

During LKM execution, VTW detects if the kernel data ci' at the address ai points to kernel code segment (⑫). If there is a ci' that does not point to the kernel code segment, it can be determined that the kernel control flow has been redirected (10 in "Detection of Rootkit Attacks"). After detecting the tampered kernel data ci', VTW reads the ci stored in Di and writes it into Di.ai (step 11). Then the tampered kernel data is restored to its initial value.

Different with kernel rootkits, the legal LKMs will neither modify the state descriptors for self hiding, nor modify the control data with system function pointing features for the purpose of control flow redirection.

Moreover, the way we protect the dynamic kernel data is to check the integrity of the protected data at specific stages of LKM execution (such as scheduling and jumps), rather than restricting all executing entities' access to the data structure to which the kernel data belongs. As a result, VTW can identify rootkits from all LKMs and will not affect the execution of other LKMs.

5.3 Bypassing Resistance of VTW Effects

Taking VTW's loading time as the demarcation point, LKMs can be divided into the loaded LKMs and the unloaded LKMs. Once VTW is successfully loaded, it can monitor and

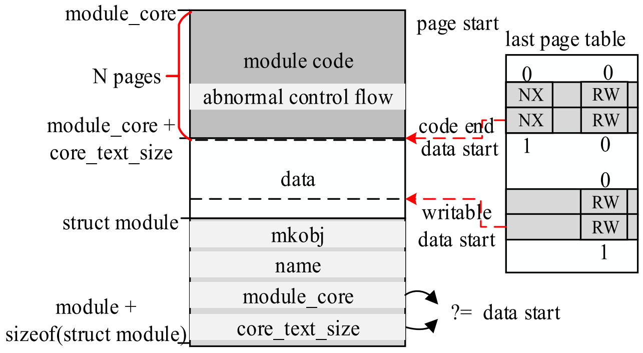


Fig. 4. Locate the hidden "struct module".

control the loading and execution of all unloaded LKMs timely, making it impossible to bypass detection.

For the LKMs that have been loaded before VTW loading, VTW cannot routinely detect and trace them in "host" mode, which increases the risk of bypassing VTW.

There are two ways to solve this problem. One is to set VTW to start at boot, so that most LKMs will be included in the monitoring scope. However, this method is still ineffective for those LKMs that also start as boot.

The second is the use of kernel integrity check method to locate the malicious LKM. The redirected control flow belongs to the rootkit. It can be determined whether the rootkit has been hidden by matching the control flow address (*mal_addr*) with every code segment (*module->core, module->core+module->core_size*) in the LKM list. If yes, *mal_addr* will be used as the starting point for locating the hidden LKM. The process is shown in Fig. 4.

The code segment and data segment of LKM are continuous in the virtual space, and their last page table entries are adjacent. Therefore, by using the executable difference between them (*NX* bit of the last page table entry), the end address (*code_end*) of the code segment and the first address (*data_start*) of the data segment can be identified.

After that, we use the read and write permissions of the data segment (the RW bit of the last page table entry) to obtain the first address (*writable_data_start*) of the writable data segment. The status descriptor (*struct module*) of the LKM is stored in the writable data segment.

According to the page alignment characteristics of the LKM code segment, the first address of the code segment is *data_start-N * page_size*, whose value is *module->module_core*. Starting from *writable_data_start*, we assume the starting address of each byte is the first address of LKM code.

Then we calculate the address of *module->core_text_size* based on the relative offset between *module->module_core* and *module->core_text_size*. The correct representation of *module->module_core* and *module->core_text_size* is that the last 12 bits are all 0, and the sum of the two is *data_start*.

Next, we check all the memory after *writable_data_start* with a single byte as the step value, until we get the eligible *module->module_core* and *module->core_text_size*. Finally, the first address of the struct module is calculated using the offset of *module->module_core* relative to the starting address of the *struct module*. Compared with the first method, this method has a wider scope, but the implementation process is relatively more complicated. VTW uses both methods to achieve the best defense effect.

TABLE 5
Required Conditions for Tracing Rootkit Effects

Operations	Condition setting
Permission setting	① $T ::= (\mathscr{L}_1, \mathscr{L}_2 \ldots \mathscr{L}_\mathrm{i} \ldots \mathscr{L}_\mathrm{c})$ $\mathscr{R} ::= (\mathscr{L}_\mathrm{c})$
	② TurnOnExe $(\mathscr{L}_\mathrm{c})$ TurnOffExe$(\mathrm{C}T\mathscr{L}_\mathrm{c})$
LKMs running on cores	③ $\mathscr{F} ::= (\mathscr{L}_1 \ldots \mathscr{L}_\mathrm{i} \mid \mathrm{i} \leq \mathrm{cpu\_counts})$
	④ $\mathscr{L}_\mathrm{j} \bowtie \mathrm{cpu\_core[k]}$
	⑤ $\mathscr{L}_1 \propto \alpha_1$ $\mathscr{L}_\mathrm{i} \propto \alpha_\mathrm{i}$ $\mathscr{L}_\mathrm{j} \propto \alpha_\mathrm{j}$
	⑥ \$$\mathscr{L}_\mathrm{j}$ & $\mathscr{F} \neq \emptyset$

6 ROOTKIT TRACING PROCESS

The kernel rootkit initiating attack may be the LKM that is being loaded or has been loaded. The participant(s) of one attack may be a single module or multiple modules.

We refer to the direct initiator of tampering with kernel data as behavior carrier. The original initiator of the attack is called action carrier. The LKM whose code replaces the original kernel function is called function carrier.

To trace all kernel rootkits participating in an attack, VTW introduces 2 new methods: "Tracing of Forward and Backward of Rootkit Attacks" and "Attack Playback Procedure". The conditions required by the methods are shown in Table 5.

Other LKMs participating in the attack are called process carrier. There may exist an overlap among these carriers. For example, the LKM being loaded can redirect the kernel control flow to the exported function. In this attack scenario, the LKM belongs to both action carrier and behavior carrier, the loaded LKM belongs to the function carrier, and there is no process carrier.

All the monitored LKMs form the tuple T, and all LKMs participating in a same event form the tuple $\mathscr{R}$ (①in Table 5). When an LKM is running, we enable its execution permission. At the same time, except for the LKMs that are running in other CPU cores, the execution permissions of all other LKMs are closed (②). All the LKMs running in different cores form a tuple $\mathscr{F}$ (③in Table 5). The LKM to be scheduled ($\mathscr{L}$j) for execution is associated with the k-th CPU core (④). The memory to which the LKM belongs is recorded as α (⑤).

6.1 Trace LKM Execution Events

There are three ways to execute a rootkit. The first one is the LKM calls its initialization function during module loading. In this type, the attack has been launched before the load is complete.

The second one is the LKM's exported function is called by other modules after it has been loaded. The loaded LKM can be called to modify the kernel data, or it can be used as a function carrier to replace the kernel's original function.

The third one is the LKM is executed by creating a kernel thread. The attack can be triggered at any time in the thread's life time.

An entire attack may cover one or several execution types of the above. To monitor the execution of LKM, VTW sets the target loaded LKMs to be unexecutable.

However, this method is less efficient due to a large number of memory setting operations. Taking an LKM with code segment size of 4MB as an example, VTW must operate EPT pagetables more than 2000 times to complete one

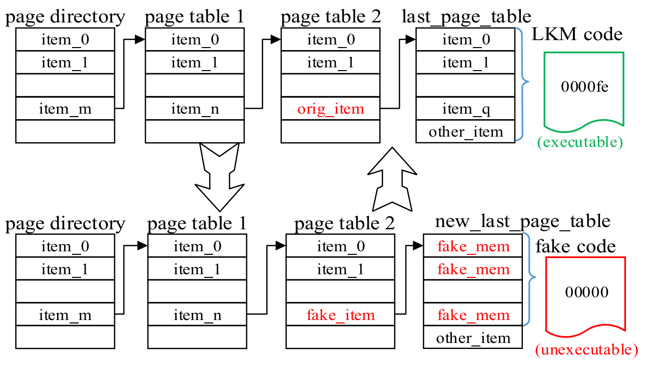


Fig. 5. The pagetable redirection mechanism.

time of execution permission opening and closing. To reduce the number of pagetable setting operations, VTW establishes a mechanism to manage LKM's executing permissions, as shown in Fig. 5.

VTW first obtains the address range of LKM code segment through "*struct module*". It then calculates the last-level pagetable (*last_page_table*) that can look up the entire address range of LKM code. The pagetable entry (*orig_item*) storing *last_page_table* address in its upper level pagetable will be recorded.

After that, VTW creates a new pagetable (*new_last_page_table*) and sets it to be "unwritable" via EPT. It overwrites *orig_item* with the address of *new_last_page_table*. Next, VTW copies all the contents of *last_page_table* into *new_last_page_table*. At the same time, it creates a memory page for each target LKM and sets them to be "unexecutable" via EPT. The memory page is called *fake_code*, and its address is *fake_mem*.

Finally, all entries in *new_last_page_table* corresponding to the LKM's code segment are filled with *fake_mem*. If we attempt to open the LKM's execution, *orig_item* will be filled into its original address. If we attempt to close the LKM's execution, *orig_item* will be rewritten with *fake_item*. Therefore, the control flow will be redirected to *fake_code*. The EPT exception address (*fake_item*) will tell which LKM is about to execute.

In LKM execution, the first LKM being executed is called the first module. The LKM that issues inter-module call request is referred to as the active module, and the LKM to be called is referred to as the called module.

Before the first LKM is executed, VTW reads the calling function's return address stored in kernel stack and sets an execution breakpoint there. Then, VTW enables the LKM's execution permission. When the control flow is switched among LKMs, we trace all LKMs through which the control flow flows. The tracing scheme is specified below.

6.2 Forward Versus Backward of Rootkit Attacks

The rootkit attacks are carried out in forward and backward phases as specified blow in 11 steps. This method is used to monitor the jump and return of the LKM control flow. Steps 1~6 monitor the jump; steps 7~11 monitor the return process.

```
1.  ConFlowTrans[$\mathscr{L}_c \rightarrow \mathscr{L}_i$]$\nsucc$SysMod[guest$\rightarrow$host]
2.  $\nsucc \mathscr{R} = \mathscr{R}$++ $\mathscr{L}_i$
3.  $\nsucc$IF CheckKernel() = = 0
4.    $\triangleright$ TurnOffExe ($\mathscr{L}_c$) TurnOnExe ($\mathscr{L}_i$)
5.  $\nsucc$ELSE HandleException($\mathscr{R}_0$)
6.    $\triangleright$ RecoverData() InjectGP()
7.  ConFlowRet[$\mathscr{L}_i \rightarrow \mathscr{L}_c$] $\nsucc$SysMod[guest$\rightarrow$host]
8.  $\nsucc$IF CheckKernel() = = 0
9.    $\triangleright$ TurnOffExe ($\mathscr{L}_i$) TurnOnExe ($\mathscr{L}_c$)
10. $\nsucc$ELSE HandleException($\mathscr{R}$)
11.   $\triangleright$ RecoverData() InjectGP()
```

In the procedure, any jump of control flow between LKMs (*ConFlowTrans*) will trigger the OS to fall into "host" (step 1) because of the permission setting of LKM. VTW records each LKM through which the control flow flows (step 2), and checks the validity of the kernel data at each jump. If the data is legal (step 3),

VTW will open the execution permission of the called module (*TurnOffExe*) and close the execution permission (*TurnOnExe*) of the active LKM (step 4). Otherwise, VTW handles the illegal LKM (steps 5~6) through the function *HandleException*, and the handling operations include data recovery (*RecoverData*) and #GP injection (*InjectGP*).

When the control flow of the LKM returns (*ConFlowRet*), VTW traces the return action (steps 7~11). When returning to the breakpoint set at the return address of the first LKM, it means the current execution ends.

When the active LKM issues a call request between LKMs or the control flow returns to the previous LKM, VTW will detect whether the current kernel data has been tampered with by the active LKM. For dynamic kernel data, VTW uses the method "Detection of Rootkit Attacks" for detection.

For the static kernel data with write protection, any tampering will cause the OS to fall into "host". Then, VTW sets the OS to single-step debugging and enables the write permission of the kernel data. Therefore, after LKM completes tampering with the static kernel data, it will fall into "host" again. Finally, VTW reads the tampered data and restore it to initial value with the backed up data.

Through the above operations, VTW can get the tampered data. Comparing the tampered data with the code segment range of all LKMs, we can get the LKM to which the redirected control flow belongs.

If it is detected that the kernel data has been tampered with, the current running module will be regarded as the behavior carrier of the attack. The first module will become the action carrier, and the module to which the redirected control flow belongs will become the function carrier.

Other LKMs will become the process carriers. To prevent further damage, VTW injects a general protection exception into the current control flow. In the end, the tampered kernel data will be recovered and the current malicious operations will be terminated.

6.3 Task Switching in Multicore Execution

When a task switch occurs during LKM execution, the LKM's CPU resources will be reclaimed. If the LKM is no longer scheduled, it will never generate inter-module call requests. The execution breakpoint we set at the return address will never be executed. Due to the lack of trigger conditions, VTW will ignore the LKM's impact on kernel after the last detection.

Different LKMs may be executed concurrently among multiple cores, which will affect the identification to the behavior carrier. For example, two LKMs may run simultaneously in different CPU cores.

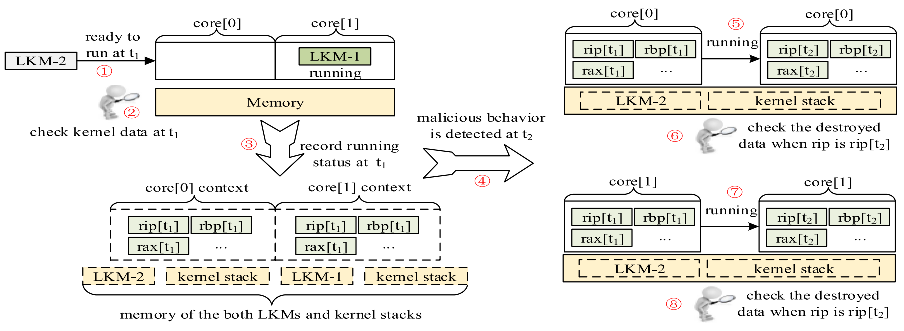


Fig. 6. Tracing task switch between two execution cores of a Linux server.

If the dynamic kernel data is detected to be tampered with during their running, we cannot figure out which of the two LKMs has committed the malicious operation.

To ensure the accurate traceability, VTW introduces an action playback method, which is shown as "Attack Playback Procedure". When an LKM is scheduled to execute and there is one or more LKMs being executed in other cores, this method will be started.

6.4 Attack Playback Procedure

In the case where multiple LKMs are running in multiple cores at the same time, this method is used to identify which LKM is the kernel rootkit. The attack playback procedure is specified below. Steps 1~5 backup the required contents.

Steps 6~9 perform the playback. When the LKM $\mathscr{L}$j is loaded to run on a core (*LoadOnCore*), the execution must trigger the OS to fall into the host mode (step 1 in "Attack Playback Procedure"). After that, $\mathscr{L}$j will be recorded into the tuple $\mathscr{F}$ (step 2).

The VTW first checks whether the current kernel data integrity is corrupted through the function *CheckKernel*. If not (step3), it will copy the current CPU's context information, the kernel stack, and the LKM data segment (all these are called playback contexts).

```
1. LoadOnCore($\mathscr{L}_j$)$\nsucc$SysMod[guest$\rightarrow$host]
2. $\mathscr{F}$++ $\mathscr{L}_j$
3. IF CheckKernel()$\neq$0
4.     $\triangleright$Case[i = = 1] $\nsucc$ Backup($\mathscr{L}_i$) Backup($\mathscr{L}_j$)
5.     $\triangleright$Case[i>1] $\nsucc$ Backup($\mathscr{L}_j$)
6. ELSE $\triangleright$Case[i = = 1] $\vdash$ $\mathscr{L}_i$ is a rootkit
7.     $\triangleright$Case[i>1] $\vdash$ $\mathscr{L}_r \in \mathscr{F}$
8.     $\triangleright$ForEach[$\mathscr{L}_m \in \mathscr{F}$] ExeBack($\mathscr{L}_m$)
9.     $\triangleright$IF CheckKernel() = = 0 $\vdash$ $\mathscr{L}_r$ is $\mathscr{L}_m$
```

If there is more than one LKM is running on different CPU cores, all the running LKMs have been backed up. So we only need to back up the LKM to be executed (step 4). If there is only one running LKM, both the LKM being executed and to be executed need to be backed up (step 5). After completing the backup, VTW switches OS to "guest" mode again to continue running.

To figure out which LKM is a kernel rootkit, the actions will be played back one by one for all LKMs running in other cores through the functions *ForEach* and *ExeBack* (step 8). The playback steps are shown in Fig. 6. The playback process is specified below in 5 steps.

1) Rewrite the current kernel stack and LKM's data with the previously backed up content.
2) Fill the "guest state" field of VMCS with the backed up CPU context and restore the corrupted kernel data.
3) Switch OS to "guest" mode to start LKM execution.
4) Stop execution at the instruction position where the kernel data is destroyed.
5) Detect whether a corrupted data will trigger a secondary destruction. The current LKM is identified as the behavior carrier. If this kernel data remains intact, then switch to resume playback from step (1).

Through the method, each LKM will be executed from the location where the kernel data has not been corrupted, and terminate at the execution location where the kernel data is corrupted. If the LKM being played back corrupts the integrity of the kernel data, it will be identified as a rootkit (step 9).

After the tracing is completed, VTW will resume the normal execution of the remaining LKMs, and let them continue execution from the position where the kernel data integrity is corrupted.

During action replay, the restored kernel data includes the tampered kernel data and the playback contexts. The recovery of the former will maintain the integrity of the kernel, and the latter is only related to the LKM to be replayed. Therefore, the recovered kernel data does not affect the execution of other LKMs.

We have analyzed 23 kernel rootkits and found that except for playback contexts, they will not modify other kernel data. During the monitoring process of the 72 normal LKMs (such as *nf_nat*, *ib_cm*, and *snd_seg*, etc.) that are frequently used in Linux, we found that normal LKMs will not modify the protected kernel data, nor will they modify anything outside the playback contexts.

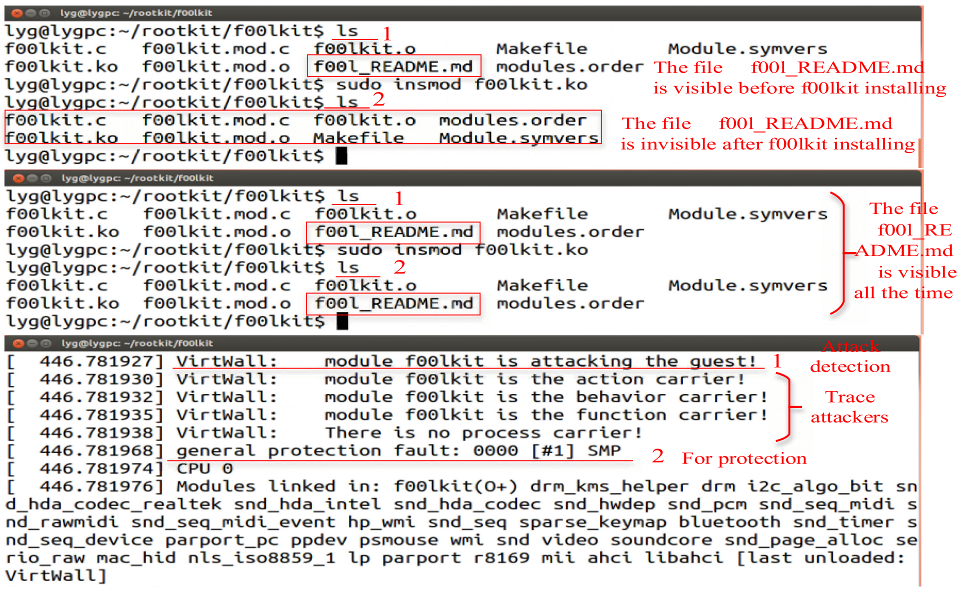
Therefore, we do not need to restore other kernel data except for the tampered kernel data and the playback contexts when performing action playback, which does not affect the correctness of other LKMs.

7 Experiments and Performance Analysis

We apply several kernel rootkits and benchmarks to test the defense effects and running efficiency of the VTW.

7.1 Experimental Environment

The physical host in the experiment is an HP desktop computer configured with an Intel i3-9100 @ 3.6 GHZ 4-core processor, 8G memory, and 256 GB hard disk. The installation environment of different rootkits is very different. To



```
lyg@lygpc:~/rootkit/f00lkit
lyg@lygpc:~/rootkit/f00lkit$ ls
f00lkit.c    f00lkit.mod.c  f00lkit.o        Makefile       Module.symvers
f00lkit.ko   f00lkit.mod.o  f00l_README.md   modules.order
lyg@lygpc:~/rootkit/f00lkit$ sudo insmod f00lkit.ko
lyg@lygpc:~/rootkit/f00lkit$ ls
f00lkit.c    f00lkit.mod.c  f00lkit.o  modules.order
f00lkit.ko   f00lkit.mod.o  Makefile   Module.symvers
lyg@lygpc:~/rootkit/f00lkit$
```

```
lyg@lygpc:~/rootkit/f00lkit
lyg@lygpc:~/rootkit/f00lkit$ ls
f00lkit.c    f00lkit.mod.c  f00lkit.o        Makefile       Module.symvers
f00lkit.ko   f00lkit.mod.o  f00l_README.md   modules.order
lyg@lygpc:~/rootkit/f00lkit$ sudo insmod f00lkit.ko
lyg@lygpc:~/rootkit/f00lkit$ ls
f00lkit.c    f00lkit.mod.c  f00lkit.o        Makefile       Module.symvers
f00lkit.ko   f00lkit.mod.o  f00l_README.md   modules.order
lyg@lygpc:~/rootkit/f00lkit$
```

```
lyg@lygpc:~/rootkit/f00lkit
[   446.781927] VirtWall:    module f00lkit is attacking the guest!
[   446.781930] VirtWall:    module f00lkit is the action carrier!
[   446.781932] VirtWall:    module f00lkit is the behavior carrier!
[   446.781935] VirtWall:    module f00lkit is the function carrier!
[   446.781938] VirtWall:    There is no process carrier!
[   446.781968] general protection fault: 0000 [#1] SMP
[   446.781974] CPU 0
[   446.781976] Modules linked in: f00lkit(O+) drm_kms_helper drm i2c_algo_bit sn
d_hda_codec_realtek snd_hda_intel snd_hda_codec snd_hwdep snd_pcm snd_seq_midi s
nd_rawmidi snd_seq_midi_event hp_wmi snd_seq sparse_keymap bluetooth snd_timer s
nd_seq_device parport_pc ppdev psmouse wmi snd video soundcore snd_page_alloc se
rio_raw mac_hid nls_iso8859_1 lp parport r8169 mii ahci libahci [last unloaded:
VirtWall]
```

1
2
The file f001_README.md is visible before f00lkit installing
The file f001_README.md is invisible after f00lkit installing
1
2
The file f001_README.md is visible all the time
1
Attack detection
Trace attackers
2
For protection

Fig. 7. Visualization of the process of a typical detected rootkit attack (f00lkit).

install the rootkit *f00lkit*, we use Ubuntu12.04 with kernel 3.2.16 as the tested OS.

7.2 Rootkit Detection, Defense, and Traceability

The *f00lkit* hides the target objects by modifying the syscall table. It redirects the control flow to its code segment. The detection effect of VTW on *f00lkit* is shown in Fig. 7.

The upper, middle and lower three windows are the intrusion performance to the OS before VTW loading, the intrusion performance to the OS after VTW loading, and the defense result of VTW respectively. In the upper window, we find that *f00lkit* can hide files prefixed with "*f00l_*", which is shown as line 2.

The middle window shows *f00lkit* no longer has a hidden effect on files prefixed with "*f00l_*" after VTW is loaded. When *f00lkit* attempts to tamper with the system call table, VTW detects its intention and generates an "EPT violation". Then VTW issues #GP to "guest" to prevent it.

To test the traceability of VTW, we rewrite the rootkits *f00lkit* and *xingyiquan*, and introduce two auxiliary LKMs *jmp_lkm* and *action_lkm*. In the attack scenario we set, *f00lkit* does not destroy kernel data. It has an exported function *func_1* that is used to replace system functions to hide files with the "*f00l_*" prefix.

There is also an exported function *func_2* in the auxiliary LKM *action_lkm*. When *func_2* is executed, it will tamper with the kernel data and redirect the system control flow to *func_1*. The exported function in auxiliary LKM *jmp_lkm* is *func_3*. The function *func_3* will call *func_2* during its execution.

The above three exported functions will not be executed until they are called by other LKMs. When *xingyiquan* executes, *func_3* is called by *xingyiquan*. At last the kernel function is redirected to *f00lkit* via *func_2*.

The LKM *action_lkm* tampers with system call table directly, and it is a behavior carrier. The original system call function is replaced by a function in *f00lkit*. Therefore, VTW determines *f00lkit* is a function carrier. The first initiator of this attack was *xingyiquan*, which makes control flow jump to other modules through *lkm_jmp*. So *xingyiquan* is determined as the action carrier, and *jmp_lkm* is determined as the process carrier.

VTW detects, defends, and traces kernel rootkits including *adore-ng*, *kbeast*, *wnps*, *brootus*, *diamorphine*, *z-rootkit*, and *suterusu*, etc. VTW sets write protection to kernel objects such as system call table through EPT pagetables. Therefore, any write operation will cause the OS to fall into the "host" mode from the "guest" mode.

Then VTW uses the #GP exception to prevent the illegal operation. The entire execution of the kernel rootkits is monitored by VTW. By manipulating the execution permissions of LKMs, VTW can capture the execution of each LKM. Therefore, the calls, jumps and returns of LKMs are recorded synchronously.

7.3 Performance Evaluation

In this section, the execution overhead of VTW on CPU is measured with *nbench* [27], the impact on system latency and bandwidth is measured by Lmbench [28], and the impact on I/O is measured by IOMeter [29]. All the results are normalized with the tests in the native OS.

Nbench Test. The test result is shown in Fig. 8. The VTW introduces a performance overhead of less than 2 percent on CPU. VTW is essentially a lightweight hypervisor and does not provide complex virtualization functions like other virtualization platforms (such as Xen). Therefore, it introduces very little performance overhead to the CPU.

Lmbench Test. Lmbench is used to measure the system latency and bandwidth. The testing results are shown in Figs. 9a, 9b, 9c. VTW increased memory and network latency by an average of 8.8 percent and file operation latency by an average of 5.9 percent. The communication bandwidth is reduced by 2.2 percent.

During the Lmbench test, we found that the OS switching frequency between "host" and "guest" increased significantly. Most of the switching is caused by the instruction *cpuid*. Because the cache operation speed is very fast, it is extremely sensitive to any delay. Therefore, the effect of

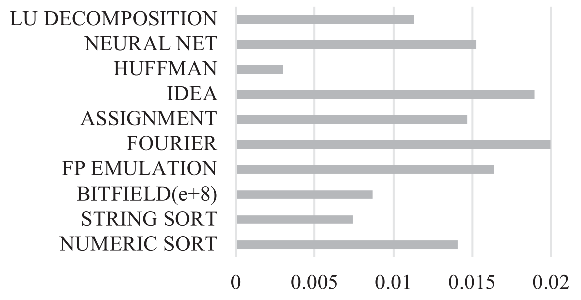


Fig. 8. Execution overhead of the VTW scheme. The *x*-axis shows the running speed loss factor, and the *y*-axis are the benchmark programs executed.

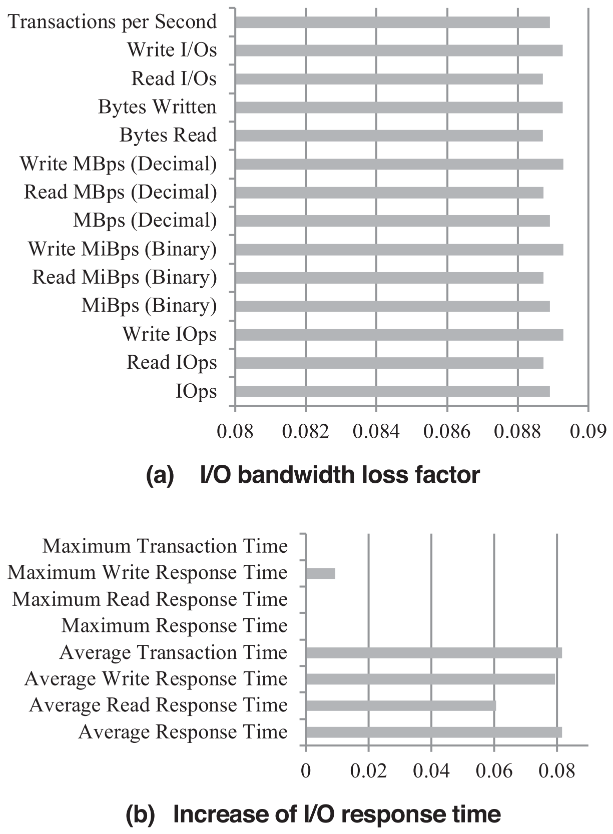


Fig. 10. I/O performance overhead plotted as the I/O bandwidth loss factor and the increase of I/O time.

mode switching on the cache is particularly obvious, resulting in a delay cost close to 20 percent.

IOMeter Test. IOMeter is a test tool developed by Intel to test the maximum disk I/O performance and maximum data throughput. The test results are shown as Figs. 10a, 10b. The figure shows that VTW causes an average I/O bandwidth reduction of 8.9 percent, and an average running time increase of 3.9 percent.

VTW's impact on I/O is mainly caused by running time of I/O operations, which reduces I/O throughput and increases I/O response time. The overhead introduced by VTW includes storage and computation overhead.

The storage overhead mainly refers to memory overhead, including memory space overhead and addressing time overhead. VTW occupies less than 200KB of memory space to store its core code and data, and 64 KB to restore the kernel data to be protected.

The calculation overhead comes from the event injection, kernel protection and exception handling introduced by VTW. These operations will cause the OS to fall into "host" from guest mode.

In host mode, VTW will take over control flow and perform exception handling. After the exception processing is completed, VTW switches the OS back to "guest" mode again, and returns control flow to the OS for continued execution.

The execution of "guest" mode is blocked throughout the process. Therefore, performance indicators such as execution speed, latency, network latency, and I/O throughput will be affected. In addition, mode switching cause the refresh of TLB and Cache [30], [31], which further increases the performance impact on the OS.

When VTW is running, the factors that cause the OS to switch modes include instruction trapping and event trapping. The former is triggered by the execution of specific instructions, while the latter is triggered by the specific events. In]guest mode, the execution of the instructions CPUID, GETTSEC, INVD, XSETBV, and all VMX instructions except VMFUNC will causes the OS to fall into "host" unconditionally.

The events that trigger OS mode switching include module loading, module uninstall, state switching during module loading, jump and return of control flow between modules, schedule execution of the module, update of the "host" private pagetable, tampering with static kernel objects, single-step debugging mode, and action playback.

Impact on the Execution Speed of LKM. In order to measure the impact of VTW on the execution speed of LKM, we introduced two test modules *LKM_1* and *LKM_2*. They

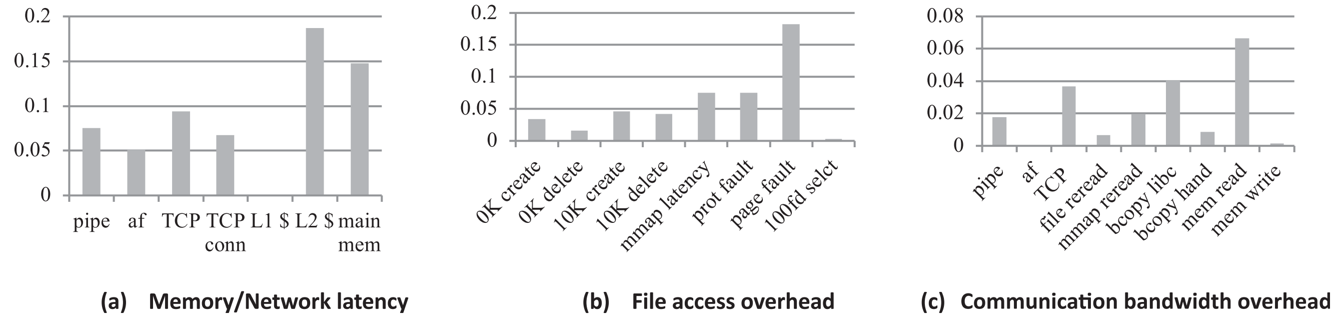


Fig. 9. Memory /network access latency, file management overhead and Input/output bandwidth reduction are plotted as the system percentage against the benchmark or network, memory, and file access operations.

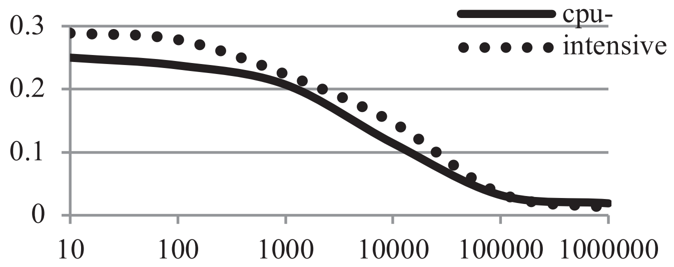


Fig. 11. Speed loss factor of the LKM plotted against the number decimal digits of π for the solid line, and the number of I/O operations for the dot line.

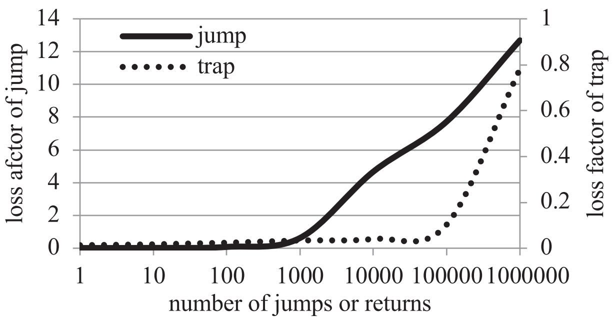


Fig. 12. The impact of system trap and control flow jump on LKM execution speed.

belong to the CPU-intensive module and I/O-intensive module respectively. The former is used to calculate the value of π, and the latter is used to read and write files.

The test results are plotted in Fig. 11. The abscissa indicates the number of digits after the π decimal point for the solid line, and the number of file operations relative for the dot line. The ordinate indicates the speed loss ratio.

On the contrary, when the module runs for a long time, the performance loss ratio caused by VTW is smaller. For normal LKMs, VTW only interferes with their execution when they are loaded and their status is updated. The execution blocking time of “guest” caused by VTW is relatively fixed. When the LKM’s execution time is short, VTW blocks the “guest” mode for a larger proportion of time, so the performance loss ratio becomes larger.

The number of traps and the number of jumps of control flow between modules are the key factors affecting the execution speed of LKM. We take an *LKM_3* with a running time of about 7.6s as the test object, and measure the impact of the number of traps on its execution speed.

We rewrite *LKM_1* and *LKM_2* so that control flow jumps between them. They can be used to measure the impact of jump times on LKM’s execution speed. The experiment results are shown as Fig. 12.

The dot line shows that VTW has a small effect on the execution speed of LKM when traps are just a few. When the trap number exceeds 1,000,000, the execution of LKM will be slowed down to 80 percent. The solid line shows that, when the jump number exceeds 100, VTW slows down to 8 percent. When the jump number exceeds 1000, the execution speed of LKM will decrease sharply.

The impact of control flow jump on LKM execution is greater than that of the traps. A complete jump and return involves 2 mode switching and 4 code execution permissions. Therefore, VTW will experience more overhead when processing low jumps between modules.

7.4 Comparison With Other Defense Schemes

In Table 6, we compare the proposed VTW performance with seven reported rootkit defense schemes. Most other rootkit defense schemes apply the introspection techniques of virtual machines, which are very different from VTW approach which is mainly based on event traceability.

We compare them qualitatively in 4 performance areas: *detection, defense, traceability and portability*. The VTW has obvious advantages in these 4 areas as agued in previous sections. More benchmark experiments are needed in future work to reveal some quantative results.

Our VTW is designed to support only Intel processors and protect only Linux-based x-86 servers. The current VTW edition does not protect servers running Windows or other OS platform. However, the Virtuoso scheme have reported with high portability across different severe platforms [14].

According to the ability to deal with malware variants, we divide the detection, defense and traceability capabilities into 3 levels: good, average, and poor.

For rootkit detection, VTW, Xtier, Virtuoso, and RTKDSM are all rated good. For defense and traceability, all methods are rated average or poor, except DIKernel has shown some defense capability. Like VTW, all reported schemes work on x86 processors exceprt for DIKernel on arm-v7.

Our experiments have revealed some measured results on CPU and storage overheads. The VTW shows distinct advantages in the system overheads to carry out all the detection and filtering operations, as compared with remaining methods in Table 6. In particular, we want to point out the excessive overheads in using VMST and PTKDSM schemes [15], [17].

TABLE 6
Comparison of VTW Scheme Against Other Defense Schemes

Defense Schemes	Detection	Defense	Traceability	CPU Overhead	Storage Overhead	Platform Protected
VTW(our model)	Good	Good	Good	<2%	>264KB	Linux
X-TIER [12]	Good	Poor	Average	<5%	14%~17%	Linux, Windos
SYRINGE [13]	Average	Average	Poor	8%	(unknown)	Windos
Virtuoso [14]	Good	Average	Poor	(unknown)	(unknown)	Linux, Windows, Haiku
VMST [15]	Average	Poor	Poor	930%	(unknown)	Linux
VMwatcher [16]	Average	Poor	Poor	<11%	(unknown)	Linux, Windows
RTKDSM [17]	Good	Poor	Poor	>110%	(unknown)	Windows
DIKernel [26]	Poor	Good	Poor	<10%	1500 code lines	Linux

8 Concluding Remarks

This paper proposes a new virtual wall (VTM) method for filtering kernel rootkit attacks on Linux servers. In summary, we have demonstrated that our VTW protection scheme has good performance In rootkit detection, defense, and traceability. VTW results in much lower memory and storage overheads than any other rootkit defense systems.

All operations that violate security policies in guest mode will cause the OS to trap into a host mode. The VTW leverages a memory access control mechanism and an event injection mechanism to perform the rootkit filtering process.

To trace the execution path of LKMs and all kernel attack participants, we propose a tracing mechanism based on the LKM execution paths. Our VTW protects the integrity of the static kernel data in real time. For dynamic kernel objects, VTW checks the validity of kernel data during the tracing process. Thus every corrupted LKM data can be detected and recovered.

Our VTW uses general protection exception to prevent further damage. When the participants of a kernel attack involve multiple kernel rootkits, we can obtain the execution entities related to the attack by checking the LKM execution paths. We trace all kernel rootkits participating in an attack. The VTW introduces an added 2 percent to the total CPU time experienced.

On the negative side, the VTM is limited to protecting Linux servers only. Our VTW scheme only supports Intel processors and Linux system. VTW does not run with AMD and ARM processors, or any servers running Windows. This is also true in the SYRINGE [13], VMST [15], and DIKernel [26] protection schemes.

The VTW has limited defense effects on Bios or user-level rootkit attacks. For the rootkits that may damage the integrity of the kernel, they may use */dev/kmem* or */dev/mem* or any other method, which could be extended from the VTW in this paper. Due to page limit, we will resort those extensions in the future work.

Acknowledgments

This research was supported in part by Key-Area Research and Development Program of Guangdong Province under contrat 2019B010155002 and in part by Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS). The research work was carried out at the AIRS Research Center for Artificial Intelligence, Cloud and IoT Edge Computing, which is directed by Prof. Kai Hwang at the Chinese University of Hong Kong, Shenzhen, China.

References

[1] K. Muthumanickam and E. Ilavarasan E, “CoPDA: Concealed process and service discovery strategy to reveal rootkit footprints,” *Malaysian J. Comput. Sci.*, vol. 28, no. 1, pp. 1–15, 2015,.

[2] E. T. Mooring and P. Yankovsky, “Systems and methods involving features of hardware virtualization: Separation kernel hypervisors, hypervisors, hypervisor guest context, hypervisor contest, rootkit detection and prevention” *Opt. Express*, vol. 22, no. 9, pp. 10398–10407, 2015.

[3] X. Wang and R. Karri, “Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 3, pp. 485–498, Mar. 2016.

[4] J. Joy, A. John, and J. Joy, “Rootkit detection mechanism: A survey,” in *Proc. Int. Conf. Parallel Distrib. Comput. Technologies Appl.*, 2011, pp. 366–374.

[5] R. Riley, “A framework for prototyping and testing data-only rootkit attacks,” *Comput. Secur.*, vol. 37, pp. 62–71, 2013.

[6] L. Catuogno and C. Galdi, “Ensuring application integrity: A survey on techniques and tools,” in *Proc. IEEE Int. Conf. Innovative Mobile Internet Services Ubiquitous Comput.*, 2015, pp. 192–199.

[7] D. Moon, B. Pan S, and I. Kim, “Host-based intrusion detection system for secure human-centric computing,” *J. Supercomputing*, vol. 72, no. 7, pp. 2520–2536, 2016.

[8] D. M. Stanley, D. Xu, and E. H. Spafford, “Improved kernel security through memory layout randomization,” in *Proc. IEEE Perform. Comput. Commun. Conf.*, 2014, pp. 1–10.

[9] A. Prakash *et al.*, “Manipulating semantic values in kernel data structures: Attack assessments and implications,” in *Proc. Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–12.

[10] J. C. Wang *et al.*, “Benefit of construct information security environment based on lightweight virtualization technology,” in *Proc. IEEE Int. Carnahan Conf. Secur. Technol.*, 2016, pp. 1–4.

[11] H. Chen and X. U. Jian, “Research on information system security protection framework based on virtualization technology,” *Electric Power Inf. Technol.*, vol. 9, no. 6, pp. 14–18, 2011.

[12] S. Vogl *et al.*, “X-TIER: Kernel module injection,” *Religious Stud.*, vol. 46, pp. 192–205, 2010.

[13] M. Carbone, M. Conover, B. Montague, and W. Lee, “Secure and robust monitoring of virtual machines through guest-assisted introspection,” *Lecture Notes Comput. Sci.*, vol. 7462, pp. 22–41, 2012.

[14] B. Dolan-Gavitt *et al.*, “Virtuoso: Narrowing the semantic gap in virtual machine introspection,” in *Proc. IEEE Symp. Secur. Privacy*, 2011, vol. 42, no. 12, pp. 297–312.

[15] Y. Fu and Z. Lin, “Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection,” in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 586–600.

[16] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection and monitoring through VMM-based “out-of-the-box” semantic view reconstruction,” *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, pp. 128–138, 2010.

[17] J. Hizver and T. C. Chiueh, “Real-time deep virtual machine introspection and its applications,” *ACM Sigplan Notices*, vol. 49, no. 7, pp. 3–14, 2014.

[18] A. S. Ibrahim *et al.*, “Supporting virtualization-aware security solutions using a systematic approach to overcome the semantic gap,” in *Proc. IEEE Int. Conf. Cloud Comput.*, 2012, pp. 836–843.

[19] X. Xu *et al.*, “Research on semantic gap problem of virtual machine,” *Wireless Pers. Commun.*, vol. 7, pp. 1–22, 2017.

[20] T. Y. Win, H. Tianfield, and Q. Mair, “Virtualization security combining mandatory access control and virtual machine IEEE introspection,” in *Proc. Int. Conf. Utility Cloud Comput.*, 2015, pp. 1004–1009.

[21] Y. Hebbal, S. Laniepce, and J. M. Menaud, “Virtual machine introspection: Techniques and applications,” in *Proc. IEEE Int. Conf. Availability Rel. Secur.*, 2015, pp. 676–685.

[22] G. Wang *et al.*, “Hypervisor introspection: A technique for evading passive virtual machine monitoring,” in *Proc. Usenix Conf. Offensive Technologies*, 2015, pp. 12–12.

[23] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2013.

[24] Q. Chen *et al.*, “Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 167–178.

[25] K. Hwang, *Cloud Computing For Machine Learning and Cognitive Applications*. Cambridge, MA, USA: MIT Press, 2017.

[26] V. J. M. Manès *et al.*, “Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions,” *Comput. Secur.*, vol. 74, pp. 130–143, 2018,.

[27] Nbench, 2020. [Online]. Available: http://linux.softpedia.com/get/System/Benchmarks/nbench-1374.shtml

[28] Lmbench, 2020. [Online]. Available: http://www.bitmover.com/lmbench/

[29] IOMeter, 2020. [Online]. Available: http://www.iometer.org/

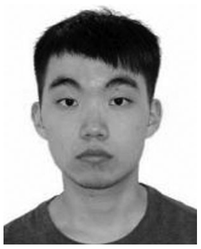
[30] W. Tang and Z. Mi, “Secure and efficient in-hypervisor memory introspection using nested virtualization,” in *Proc. IEEE Symp. Serv.-Oriented Syst. Eng.*, 2018, pp. 186–191.

[31] X. Wang, Y. Qi, Z. Wang, Y. Chen, and Y. Zhou, “Design and implementation of SecPod, a framework for virtualization-based security systems,” *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 1, pp. 44–57, Jan.-Feb. 2019.

Yong-Gang Li received the PhD degree from the University of Science and Technology of China (USTC), in 2019. Currently, he is a postdoctoral fellow with the Chinese University of Hong Kong, Shenzhen, and USTC. He is also a visiting assistant researcher with the Shenzhen Institute of Artificial Intelligence and Robotics for Society. His research interests include computer architecture, virtualization principle, cloud computing, container service, and system security.

Yeh-Ching Chung received the PhD degree in computer and information science from Syracuse University, in 1992. From 2002 to 2016, he was with the Department of Computer Science at National Tsing Hua University as a full professor. From 2003 to 2012, he served as the deputy director of Library, where he established the first UHF RFID library system in Taiwan. Currently, he is a full professor with the School of Data Science (SDS) of Chinese University of Hong Kong in Shenzhen. His research interests include parallel and distributed processing, cloud computing, big data, and embedded system.

Kai Hwang (Life Fellow, IEEE) received the PhD degree in EECS from UC Berkeley. He is a presidential chair professor with the Chinese University of Hong Kong (CUHK), Shenzhen, where he heads the AIRS Research Center for AI Cloud and IoT Edge Computing. He has published 10 scientific books and more than 260 original papers. He has received the Outstanding Achievement Award from Computer Federation of China (CFC) in 2005, and the Lifetime Achievement Award, IEEE CloudCom, in 2012.



Yuejin Li received the BS degree in financial engineering from Wuhan University, in 2016, and the two MS degrees from the London School of Economics and Political Science, and University of St. Andrews, in 2017 and 2018, respectively. Currently, he is working toward the PhD degree in computer science at the Chinese University of Hong Kong, Shenzhen. His research interests include cloud computing and financial big data analytics.