Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing

Ryan Riley[1], Xuxian Jiang[2], and Dongyan Xu[1]

[1] CERIAS and Department of Computer Science, Purdue University
{rileyrd,dxu}@cs.purdue.edu
[2] Department of Computer Science, North Carolina State University
jiang@cs.ncsu.edu

Abstract. Kernel rootkits pose a significant threat to computer systems as they run at the highest privilege level and have unrestricted access to the resources of their victims. Many current efforts in kernel rootkit defense focus on the *detection* of kernel rootkits – after a rootkit attack has taken place, while the smaller number of efforts in kernel rootkit *prevention* exhibit limitations in their capability or deployability. In this paper we present a kernel rootkit prevention system called NICKLE which addresses a common, fundamental characteristic of most kernel rootkits: the need for executing their own kernel code. NICKLE is a lightweight, virtual machine monitor (VMM) based system that transparently prevents unauthorized kernel code execution for unmodified commodity (guest) OSes. NICKLE is based on a new scheme called *memory shadowing*, wherein the trusted VMM maintains a shadow physical memory for a running VM and performs real-time kernel code authentication so that only authenticated kernel code will be stored in the shadow memory. Further, NICKLE transparently routes guest kernel instruction fetches to the shadow memory at runtime. By doing so, NICKLE guarantees that only the authenticated kernel code will be executed, foiling the kernel rootkit's attempt to strike in the first place. We have implemented NICKLE in three VMM platforms: QEMU+KQEMU, VirtualBox, and VMware Workstation. Our experiments with 23 real-world kernel rootkits targeting the Linux or Windows OSes demonstrate NICKLE's effectiveness. Furthermore, our performance evaluation shows that NICKLE introduces small overhead to the VMM platform.

1 Introduction

Kernel-level rootkits have proven to be a formidable threat to computer systems: By subverting the operating system (OS) kernel, a kernel rootkit embeds itself into the compromised kernel and stealthily inflicts damages with full, unrestricted access to the system's resources. Effectively omnipotent in the compromised systems, kernel rootkits have increasingly been used by attackers to hide their presence and prolong their control over their victims.

There have been a number of recent efforts in mitigating the threat of kernel rootkits and they can mainly be classified into two categories: (1) detecting the presence of kernel rootkits in a system [1, 2, 3, 4, 5] and (2) preventing the compromise of OS kernel integrity [6, 7]. In the first category, Copilot [4] proposes the use of a separate PCI card to periodically grab the memory image of a running OS kernel and analyze it to determine if the kernel has been compromised. The work which follows up Copilot [2] further extends that capability by detecting the violation of kernel integrity using semantic specifications of static and dynamic kernel data. SBCFI [3] reports violations of the kernel's control flow integrity using the kernel's control-flow graph. One common attribute of approaches in this category is the *detection* of a kernel rootkit's presence based on certain symptoms exhibited by the kernel *after* the kernel rootkit has already struck. As a result, these approaches are, by design, not capable of *preventing kernel rootkit execution in the first place.*

In the second category, Livewire [6], based on a virtual machine monitor (VMM), aims at protecting the guest OS kernel code and critical kernel data structures from being modified. However, without modifying the original kernel code, an attacker may choose to load malicious rootkit code into the kernel space by either exploiting kernel vulnerabilities or leveraging certain kernel features (e.g., loadable kernel module support in modern OSes). More recently, SecVisor [7] is proposed as a hypervisor-based solution to enforce the W$\oplus$X property of memory pages of the guest machine, with the goal of preventing unauthorized code from running with kernel-level privileges. SecVisor requires modifying kernel source code and needs the latest hardware-based virtualization support and thus does not support closed-source OSes or legacy hardware platforms. Moreover, SecVisor is not able to function if the OS kernel has *mixed pages that contain both code and data.* Unfortunately, such mixed kernel pages do exist in modern OSes (e.g., Linux and Windows as shown in Section 2.2).

To complement the existing approaches, we present NICKLE ("No Instruction Creeping into Kernel Level Executed")[1], a lightweight, VMM-based system that provides an important guarantee in kernel rootkit prevention: *No unauthorized code can be executed at the kernel level.* NICKLE achieves this guarantee on top of legacy hardware and without requiring guest OS kernel modification. As such, NICKLE is readily deployable to protect unmodified guest OSes (e.g., Fedora Core 3/4/5 and Windows 2K/XP) against kernel rootkits. NICKLE is based on observing a common, fundamental characteristic of most modern kernel rootkits: their ability to execute unauthorized instructions at the kernel level. By removing this ability, NICKLE significantly raises the bar for successfully launching kernel rootkit attacks.

To achieve the "NICKLE" guarantee, we first observe that a kernel rootkit is able to access the entire physical address space of the victim machine. This observation inspires us to impose restricted access to the instructions in the kernel space: only *authenticated* kernel instructions can be fetched for execution. Obviously, such a restriction cannot be enforced by the OS kernel itself. Instead,

[1] With a slight abuse of terms, we use NICKLE to denote both the system itself and the guarantee achieved by the system – when used in quotation marks.

a natural strategy is to enforce such memory access restriction using the VMM, which is at a privilege level higher than that of the (guest) OS kernel.

Our main challenge is to realize the above VMM-level kernel instruction fetch restriction in a guest-transparent, real-time, and efficient manner. An intuitive approach would be to impose W⊕X on kernel memory pages to protect existing kernel code and prevent the execution of injected kernel code. However, due to the existence of mixed kernel pages in commodity OSes, this approach is not viable for guest-transparent protection. To address that, we propose a VMM-based *memory shadowing* scheme for NICKLE that will work in the face of mixed kernel pages. More specifically, for a virtual machine (VM), the VMM creates two distinct physical memory regions: a *standard memory* and a *shadow memory*. The VMM enforces that the guest OS kernel cannot access the shadow memory. Upon the VM's startup, the VMM performs kernel code authentication and dynamically copies authenticated kernel instructions from the standard memory to the shadow memory. At runtime, any instruction executed in the kernel space must be fetched from the shadow memory instead of from the standard memory. To enforce this while maintaining guest transparency, a lightweight *guest memory access indirection* mechanism is added to the VMM. As such, a kernel rootkit will never be able to execute any of its own code as the code injected into the kernel space will not be able to reach the shadow memory.

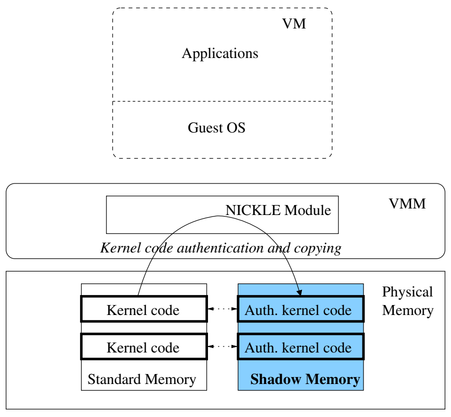
We have implemented NICKLE in three VMMs: QEMU[8] with the KQEMU accelerator, VirtualBox [9], and VMware Workstation. Our evaluation results show that NICKLE incurs a reasonable impact on the VMM platform (e.g., 1.01% on QEMU+KQEMU and 5.45% on VirtualBox when running UnixBench). NICKLE is shown capable of transparently protecting a variety of commodity OSes, including RedHat 8.0 (Linux 2.4.18 kernel), Fedora Core 3 (Linux 2.6.15 kernel), Windows 2000, and Windows XP. Our results show that NICKLE is able to prevent and gracefully respond to 23 real-world kernel rootkits targeting the above OSes, without requiring details of rootkit attack vectors. Finally, our porting experience indicates that the NICKLE design is generic and realizable in a variety of VMMs.

2 NICKLE Design

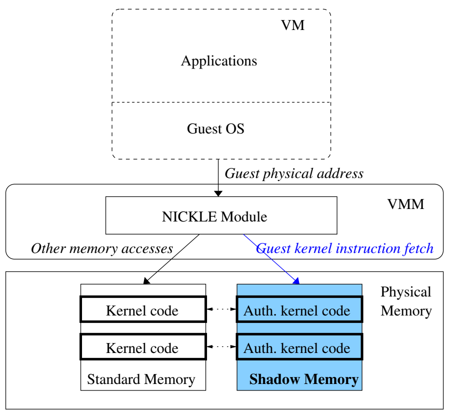
2.1 Design Goals and Threat Model

Goals and Challenges. NICKLE has the following three main design goals:

First, as its name indicates, NICKLE should prevent any unauthorized code from being executed in the kernel space of the protected VM. The challenges of realizing this goal come from the real-time requirement of prevention as well as from the requirement that the guest OS kernel should not be trusted to initiate any task of the prevention – the latter requirement is justified by the kernel rootkit's highest privilege level inside the VM and the possible existence of zero-day vulnerabilities inside the guest OS kernel. NICKLE overcomes these challenges using the VMM-based memory shadowing scheme (Section 2.2). We



(a) Kernel code authorization and copying



(b) Guest physical address redirection

Fig. 1. Memory shadowing scheme in NICKLE

note that the scope of NICKLE is focused on preventing unauthorized kernel code execution. The prevention of other types of attacks (e.g., data-only attacks) is a non-goal and related solutions will be discussed in Section 5.

Second, NICKLE should not require modifications to the guest OS kernel. This allows commodity OSes to be supported “as is” without recompilation and reinstallation. Correspondingly, the challenge in realizing this goal is to make the memory shadowing scheme transparent to the VM with respect to both the VM's function and performance.

Third, the design of NICKLE should be generically portable to a range of VMMs. Given this, the challenge is to ensure that NICKLE has a small footprint within the VMM and remains lightweight with respect to performance impact. In this paper we focus on supporting NICKLE in software VMMs. However, we expect that the exploitation of recent hardware-based virtualization extensions [10, 11] will improve NICKLE's performance even further.

In addition, it is also desirable that NICKLE facilitate various flexible response mechanisms to be activated upon the detection of an unauthorized kernel code execution attempt. A flexible response, for example, is to cause only the offending process to fail without stopping the rest of the OS. The challenge in realizing this is to initiate flexible responses entirely from outside the protected VM and minimize the side-effects on the running OS.

Threat Model and System Assumption. We assume the following adversary model when designing NICKLE: (1) The kernel rootkit has the highest privilege level inside the victim VM (e.g., the *root* privilege in a UNIX system); (2) The kernel rootkit has full access to the VM's memory space (e.g., through `/dev/mem` in Linux); (3) The kernel rootkit aims at stealthily maintaining and hiding its presence in the VM and to do so, the rootkit will need to execute its own (malicious) code in the kernel space. We note that such a need exists in most kernel rootkits today, and we will discuss possible exceptions in Section 5.

Meanwhile, we assume a trusted VMM that provides VM isolation. This assumption is shared by many other VMM-based security research efforts [1, 6, 12, 13, 14, 15]. We will discuss possible attacks (e.g., VM fingerprinting) in Section 5. With this assumption, we consider the threat from DMA attacks launched from physical hosts outside of the scope of this work.[2]

2.2 Enabling Scheme and Techniques

Memory Shadowing. The memory shadowing scheme enforces the "NICKLE" property: For a VM, apart from its standard physical memory space, the VMM also allocates a separate physical memory region as the VM's *shadow memory* (Figure 1) which is transparent to the VM and controlled by the VMM. Upon the startup of the VM's OS, all known-good, authenticated guest kernel instructions will be copied from the VM's standard memory to the shadow memory (Figure 1(a)). At runtime, when the VM is about to execute a kernel instruction, the VMM will transparently redirect the kernel instruction fetch to the shadow memory (Figure 1(b)). All other memory accesses (to user code, user data, and kernel data) will proceed unhindered in the standard memory.

The memory shadowing scheme is motivated by the observation that modern computers define a single memory space for all code – both kernel code and user code – and data. With the VMM running at a higher privilege level, we can now "shadow" the guest kernel code space with elevated (VMM-level) privileges to ensure that the guest OS kernel itself cannot access the shadowed kernel code space containg the authenticated kernel instructions. By doing so, even if a kernel rootkit is able to inject its own code into the VM's standard memory, the VMM will ensure that the malicious code never gets copied over to the shadow memory. Moreover, an attempt to execute the malicious code can be caught immediately due to the inconsistency between the standard and shadow memory contents.

The astute reader may be asking "How is NICKLE functionally different from W⊕X?" In essence, W⊕X is a scheme that enforces the property, "A given memory page will never be both writable and executable at the same time." The basic premise behind this scheme is that if a page cannot be written to and later executed from, code injection becomes impossible. There are two main reasons why this scheme is not adequate for stopping kernel level rootkits:

First, W⊕X is not able to protect mixed kernel pages with both code and data, which do exist in current OSes. As a specific example, in a Fedora Core 3 VM (with the 32-bit 2.6.15 kernel and the NX protection), the Linux kernel stores the main static kernel text in memory range $[0xc0100000, 0xc02dea50]$ and keeps the system call table starting from virtual address $0xc02e04a0$. Notice that the Linux kernel uses a large page size ($2MB$) to manage the physical memory,[3] which means that the first two kernel pages cover memory ranges

[2] There exists another type of DMA attack that is initiated from within a guest VM. However, since the VMM itself virtualizes or mediates the guest DMA operations, NICKLE can be easily extended to intercede and block them.

[3] If the NX protection is disabled, those kernel pages containing static kernel text will be of $4MB$ in size.

$[0xc0000000, 0xc0200000)$ and $[0xc0200000, 0xc0400000)$, respectively. As a result, the second kernel page contains both code and data, and thus must be marked both writable and executable – This conflicts with the W⊕X scheme. Mixed pages also exist for accommodating the code and data of Linux loadable kernel modules (LKMs) – an example will be shown in Section 4.1. For the Windows XP kernel (with SP2), our investigation has confirmed the existence of mixed pages as well [16]. On the other hand, NICKLE is able to protect mixed pages.[4]

Second, W⊕X assumes only one execution privilege level while kernel rootkit prevention requires further distinction between user and kernel code pages. For example, a page may be set executable in user mode but non-executable in kernel mode. In other words, the sort of permission desired is not W⊕X, but W⊕KX (i.e. not writable and kernel-executable at the same time.) Still, we point out that the enforcement of W⊕KX is *not* effective for mixed kernel pages and, regardless, not obvious to construct on current processors that do not allow such fine-grained memory permissions.

Another question that may be asked is, "Why adopt memory shadowing when one could simply guard kernel code by keeping track of the ranges of valid kernel code addresses ?" Indeed, NICKLE is guided by the principle of kernel code guarding, but does so differently from the brute-force approach of tracking/checking kernel code address ranges – mainly for performance reasons. More specifically, the brute-force approach could store the address ranges of valid kernel code in a data structure (e.g., tree) with $O(logN)$ search time. On the other hand, memory shadowing allows us to locate the valid kernel instruction in the shadow memory in $O(1)$ time thus significantly reducing the processing overhead. In addition, memory shadowing makes it convenient to compare the instructions in the shadow memory to those in the standard memory. If they differ (indicating malicious kernel code injection or modification), a number of response actions can be implemented based on the difference (details in Section 3).

Guest Memory Access Indirection. To realize the guest memory shadowing scheme, two issues need to be resolved. First, how does NICKLE fill up the guest shadow memory with authenticated kernel code? Second, how does NICKLE fetch authenticated kernel instructions for execution while detecting and preventing any attempt to execute unauthorized code in the kernel space? We note that our solutions have to be transparent to the guest OS (and thus to the kernel rootkits). We now present the guest memory access indirection technique to address these issues.

[4] We also considered the option of eliminating mixed kernel pages. However, doing so would require kernel source code modification, which conflicts with our second design goal. Even given source code access, mixed page elimination is still a complex task (more than just page-aligning data). In fact, a kernel configuration option with a similar purpose exists in the latest Linux kernel (version 2.6.23). But after we enabled the option, we still found more than 700 mixed kernel pages. NICKLE instead simply avoids such complexity and works even with mixed kernel pages.

Guest memory access indirection is performed between the VM and its memory (standard and shadow) by a thin NICKLE module inside the VMM. It has two main functions, kernel code authentication and copying at VM startup and upon kernel module loading as well as guest physical address redirection at runtime (Figure 1).

Kernel Code Authentication and Copying. To fill up the shadow memory with authenticated kernel instructions, the NICKLE module inside the VMM needs to first determine the accurate timing for kernel code authentication and copying. To better articulate the problem, we will use the Linux kernel as an example. There are two specific situations throughout the VM's lifetime when kernel code needs to be authorized and shadowed: One at the VM's startup and one upon the loading/unloading of loadable kernel modules (LKMs). When the VM is starting up, the guest's shadow memory is empty. The kernel bootstrap code then decompresses the kernel. Right after the decompression and before any processes are executed, NICKLE will use a cryptographic hash to verify the integrity of the kernel code (this is very similar to level 4 in the secure bootstrap procedure [17]) and then copy the authenticated kernel code from the standard memory into the shadow memory (Figure 1(a)). As such, the protected VM will start with a known clean kernel.

The LKM support in modern OSes complicates our design. From NICKLE's perspective, LKMs are considered injected kernel code and thus need to be authenticated and shadowed before their execution. The challenge for NICKLE is to *externally* monitor the guest OS and detect the kernel module loading/unloading events in real-time. NICKLE achieves this by leveraging our earlier work on non-intrusive VM monitoring and semantic event reconstruction [1, 14]. When NICKLE detects the loading of a new kernel module, it intercepts the VM's execution and performs kernel module code authentication and shadowing. The authentication is performed by taking a cryptographic hash of the kernel module's code segment and comparing it with a known correct value, which is computed a priori off-line and provided by the administrator or distribution maintainer.[5] If the hash values don't match, the kernel module's code will not be copied to the shadow memory.

Through kernel code authentication and copying, only authenticated kernel code will be loaded into the shadow memory, thus blocking the copying of malicious kernel rootkit code or any other code injected by exploiting kernel vulnerabilities, including zero-day vulnerabilities. It is important to note that neither kernel startup hashing nor kernel module hashing assumes trust in the guest OS. Should the guest OS fail to cooperate, *no* code will be copied to the shadow memory, and any execution attempts from that code will be detected and refused.

Guest Physical Address Redirection. At runtime, the NICKLE module inside the VMM intercepts the memory accesses of the VM *after* the "guest virtual address $\rightarrow$ guest physical address" translation. As such, NICKLE does not interfere

[5] We have developed an off-line kernel module profiler that, given a legitimate kernel module, will compute the corresponding hash value (Section 3.1).

with – and is therefore transparent to – the guest OS's memory access handling procedure and virtual memory mappings. Instead, it takes the guest physical address, determines the type of the memory access (kernel, user; code, data; etc.), and routes it to either the standard or shadow memory (Figure 1(b)).

We point out that the interception of VM memory accesses can be provided by existing VMMs (e.g., QEMU+KQEMU, VirtualBox, and VMware). NICKLE builds on this interception capability by adding the guest physical address redirection logic. First, using a simple method to check the current privilege level of the processor, NICKLE determines whether the current instruction fetch is for kernel code or for user code: If the processor is in supervisor mode (CPL=0 on x86), we infer that the fetch is for kernel code and NICKLE will verify and route the instruction fetch to the shadow memory. Otherwise, the processor is in user mode and NICKLE will route the instruction fetch to the standard memory. Data accesses of either type are always routed to the standard memory.

One might object that an attacker may strive to ensure that his injected kernel code will run when the processor is in user mode. However, this creates a significant challenge wherein the attacker would have to fundamentally change a running kernel to operate in both supervisor and user mode *without changing any existing kernel code*. The authors do not consider such a rootkit to be a possibility without a severe loss of rootkit functionality.

Flexible Responses to Unauthorized Kernel Code Execution Attempts If an unauthorized execution attempt is detected, a natural follow-up question is, "How should NICKLE respond to an attempt to execute an unauthenticated kernel instruction?" Given that NICKLE sits between the VM and its memory and has a higher privilege level than the guest OS, it possesses a wide range of options and capabilities to respond. We describe two response modes facilitated by the current NICKLE system.

Rewrite mode: NICKLE will dynamically rewrite the malicious kernel code with code of its own. The response code can range from OS-specific error handling code to a well-crafted payload designed to clean up the impact of a rootkit installation attempt. Note that this mode may require an understanding of the guest OS to ensure that valid, sensible code is returned.

Break mode: NICKLE will take no action and route the instruction fetch to the *shadow memory*. In the case where the attacker only modifies the original kernel code, this mode will lead to the execution of the original code – a desirable situation. However, in the case where *new* code is injected into the kernel, this mode will lead to an instruction fetch from presumably null content (containing 0s) in the shadow memory. As such, break mode prevents malicious kernel code execution but may or may not be graceful depending on how the OS handles invalid code execution faults.

3 NICKLE Implementation

To validate the portability of the NICKLE design, we have implemented NICKLE in three VMMs: QEMU+KQEMU [8], VirtualBox [9], and VMware

Workstation[6]. Since the open-source QEMU+KQEMU is the VMM platform where we first implemented NICKLE, we use it as the representative VMM to describe our implementation details. For most of this section, we choose RedHat 8.0 as the default guest OS. We will also discuss the limitations of our current prototype in supporting Windows guest OSes.

3.1 Memory Shadowing and Guest Memory Access Indirection

To implement memory shadowing, we have considered two options: (1) NICKLE could interfere as instructions are executed; or (2) NICKLE could interfere when instructions are dynamically translated. Note that dynamic instruction translation is a key technique behind existing software-based VMMs, which transparently translates guest machine code into native code that will run in the physical host. We favor the second option for performance reasons: By being part of the translator, NICKLE can take advantage of the fact that translated code blocks are cached. In QEMU+KQEMU, for example, guest kernel instructions are grouped into "blocks" and are dynamically translated at runtime. After a block of code is translated, it is stored in a cache to make it available for future execution. In terms of NICKLE, this means that if we intercede during code translation we need not intercede as often as we would if we did so during code execution, resulting in a smaller impact on system performance.

The pseudo-code for memory shadowing and guest memory access indirection is shown in Algorithm 1. Given the guest physical address of an instruction to be executed by the VM, NICKLE first checks the current privilege level of the processor (CPL). If the processor is in supervisor mode, NICKLE knows that it is executing in kernel mode. Using the guest physical address, NICKLE compares the content of the standard and shadow memories to determine whether the kernel instruction to be executed is already in the shadow memory (namely has been authenticated). If so, the kernel instruction is allowed to be fetched, translated, and executed. If not, NICKLE will determine if the guest OS kernel is being bootstrapped or a kernel module is being loaded. If either is the case, the corresponding kernel text or kernel module code will be authenticated and, if successful, shadowed into the shadow memory. Otherwise, NICKLE detects an attempt to execute an unauthorized instruction in the kernel space and prevents it by executing our response to the attempt.

In Algorithm 1, the way to determine whether the guest OS kernel is being bootstrapped or a kernel module is being loaded requires OS-specific knowledge. Using the Linux 2.4 kernel as an example, when the kernel's *startup_32* function, located at physical address `0x00100000` or virtual address `0xc0100000` as shown in the *System.map* file, is to be executed, we know that this is the first

[6] We acknowledge the VMware Academic Program for providing the source code. Due to space and licensing constraints, however, the VMware port is not further discussed or evaluated in this work. Some additional discussion of the port is available in our technical report [16].

Algorithm 1. Algorithm for Memory Shadowing and Guest Memory Access Indirection

```
Input: (1) GuestPA: guest physical address of instruction to be executed; (2) ShadowMEM[]:
       shadow memory; (3) StandardMEM[]: standard memory
1 if !IsUserMode(vcpu) AND ShadowMEM[GuestPA] != StandardMEM[GuestPA] then
2     if (kernel is being bootstrapped) OR (module is being loaded) then
3         Authenticate and shadow code;
4     else
5         Unauthorized execution attempt - Execute response;
6     end
7 end
8 Fetch, translate, and cache code;
```

instruction executed to load the kernel and we can intercede appropriately. For kernel module loading, there is a specific system call to handle that. As such, the NICKLE module inside the VMM can intercept the system call and perform kernel module authentication and shadowing right before the module-specific *init_module* routine is executed.

In our implementation, the loading of LKMs requires special handling. More specifically, providing a hash of a kernel module's code space ends up being slightly complicated in practice. This is due to the fact that kernel modules are dynamically relocatable and hence some portions of the kernel module's code space may be modified by the module loading function. Accordingly, the cryptographic hash of a loaded kernel module will be different depending on where it is relocated to. To solve this problem, we perform an off-line, a priori profiling of the legitimate kernel module binaries. For each known good module we calculate the cryptographic hash by excluding the portions of the module that will be changed during relocation. In addition, we store a list of bytes affected by relocation so that the same procedure can be repeated by NICKLE during runtime hash evaluation of the same module.

We point out that although the implementation of NICKLE requires certain guest OS-specific information, it does *not* require modifications to the guest OS itself. Still, for a closed-source guest OS (e.g., Windows), lack of information about kernel bootstrapping and dynamic kernel code loading may lead to certain limitations. For example, not knowing the timing and "signature" of dynamic (legal) kernel code loading events in Windows, the current implementation of NICKLE relies on the administrator to designate a time instance when all authorized Windows kernel code has been loaded into the standard memory. Not knowing the exact locations of the kernel code, NICKLE traverses the shadow page table and copies those executable pages located in the kernel space from the standard memory to the shadow memory, hence creating a "gold standard" to compare future kernel code execution against. From this time on, NICKLE can transparently protect the Windows OS kernel from executing any unauthorized kernel code. Moreover, this limited implementation can be made complete when the relevant information becomes available through vendor disclosure or reverse engineering.

3.2 Flexible Response

In response to an attempt to execute an unauthorized instruction in the kernel space, NICKLE provides two response modes. Our initial implementation of NICKLE simply re-routes the instruction fetch to the shadow memory for a string of zeros (break mode). As to be shown in our experiments, this produces some interesting outcomes: a Linux guest OS would react to this by triggering a kernel fault and terminating the offending process. Windows, on the other hand, reacts to the NICKLE response by immediately halting with a blue screen – a less graceful outcome.

In search of a more flexible response mode, we find that by rewriting the offending instructions at runtime (rewrite mode), NICKLE can respond in a less disruptive way. We also observe that most kernel rootkits analyzed behave the following way: They first insert a new chunk of malicious code into the kernel space; then they somehow ensure their code is `call`'d as a function. With this observation, we let NICKLE dynamically replace the code with `return -1;`, which in assembly is: `mov $0xffffffff, %eax; ret`. The main kernel text or the kernel module loading process will interpret this as an error and gracefully handle it: Our experiments with Windows 2K/XP, Linux 2.4, and Linux 2.6 guest OSes all confirm that NICKLE's rewrite mode is able to handle the malicious kernel code execution attempt by triggering the OS to terminate the offending process without causing a fault in the OS.

3.3 Porting Experience

We have experienced no major difficulty in porting NICKLE to other VMMs. The NICKLE implementations in both VMMs are lightweight: The SLOC (source lines of code) added to implement NICKLE in QEMU+KQEMU, VirtualBox, and VMware Workstation are 853, 762, and 1181 respectively. As mentioned earlier, we first implemented NICKLE in QEMU+KQEMU. It then took less than one week for one person to get NICKLE functional in VirtualBox 1.5.0 OSE, details of which can be found in our technical report [16].

4 NICKLE Evaluation

4.1 Effectiveness Against Kernel Rootkits

We have evaluated the effectiveness of NICKLE with 23 real-world kernel rootkits. They consist of nine Linux 2.4 rootkits, seven Linux 2.6 rootkits, and seven Windows rootkits[7] that can infect Windows 2000 and/or XP. The selected rootkits cover the main attack platforms and attack vectors thus providing a good representation of the state-of-the-art kernel rootkit technology. Table 1 shows

[7] There is a Windows rootkit named hxdef or Hacker Defender, which is usually classified as a user-level rootkit. However, since hxdef contains a device driver which will be loaded into the kernel, we consider it a kernel rootkit in this paper.

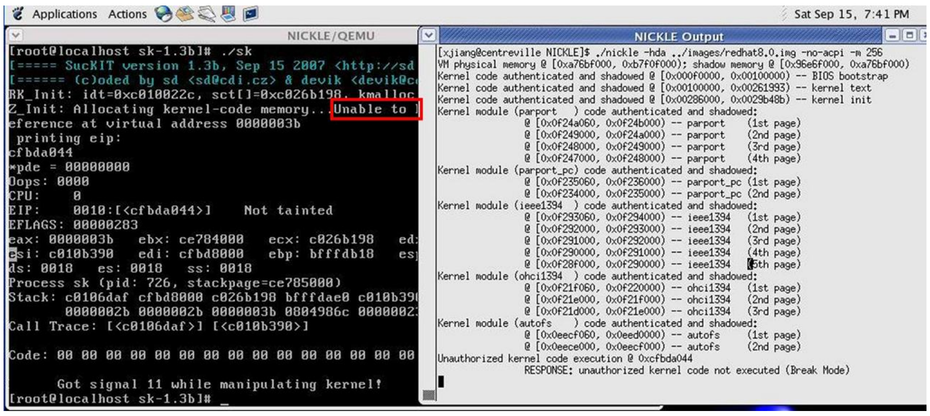
Table 1. Effectiveness of NICKLE in detecting and preventing 23 real-world kernel rootkits (*DKOM*† is a common rootkit technique which directly manipulates kernel objects; *"partial"*‡ means the in-kernel component of the Hacker Defender rootkit fails; *BSOD*§ stands for "Blue Screen Of Death")

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response			
			Rewrite Mode		Break Mode	
			Prevented?	Outcome	Prevented?	Outcome
Linux 2.4	adore 0.42, 0.53	LKM	✓	`insmod` fails	✓	Seg. fault
	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	knark	LKM	✓	`insmod` fails	✓	Seg. fault
	rkit 1.01	LKM	✓	`insmod` fails	✓	Seg. fault
	kbdv3	LKM	✓	`insmod` fails	✓	Seg. fault
	allroot	LKM	✓	`insmod` fails	✓	Seg. fault
	rial	LKM	✓	`insmod` fails	✓	Seg. fault
	Phantasmagoria	LKM	✓	`insmod` fails	✓	Seg. fault
	SucKIT 1.3b	`/dev/kmem`	✓	Installation fails silently	✓	Seg. fault
Linux 2.6	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	eNYeLKM v1.2	LKM	✓	`insmod` fails	✓	Seg. fault
	sk2rc2	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	superkit	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	mood-nt 2.3	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	override	LKM	✓	`insmod` fails	✓	Seg. fault
	Phalanx b6	`/dev/mem`	✓	Installation crashes	✓	Seg. fault
Windows 2K/XP	FU	DKOM†	✓	Driver loading fails	✓	BSOD§
	FUTo	DKOM	✓	Driver loading fails	✓	BSOD
	he4hook 215b6	Driver	✓	Driver loading fails	✓	BSOD
	hxdef 1.0.0 revisited	Driver	partial‡	Driver loading fails	✓	BSOD
	hkdoor11	Driver	✓	Driver loading fails	✓	BSOD
	yyt_hac	Driver	✓	Driver loading fails	✓	BSOD
	NT Rootkit	Driver	✓	Driver loading fails	✓	BSOD

our experimental results: NICKLE is able to detect and prevent the execution of malicious kernel code in *all* experiments using both rewrite and break response modes. Finally, we note that NICKLE in all three VMMs is able to achieve the same results. In the following, we present details of two representative experiments. Some additional experiments are presented in [16].

SucKIT Rootkit Experiment. The SucKIT rootkit [18] for Linux 2.4 infects the Linux kernel by directly modifying the kernel through the `/dev/kmem` interface. During installation SucKIT first allocates memory within the kernel, injects its code into the allocated memory, and then causes the code to run as a function. Figure 2 shows NICKLE preventing the SucKIT installation. The window on the left shows the VM running RedHat 8.0 (with 2.4.18 kernel), while the window on the right shows the NICKLE output. Inside the VM, one can see that the SucKIT installation program fails and returns an error message "*Unable to handle kernel NULL pointer dereference*". This occurs because NICKLE (operating in break mode) foils the execution of injected kernel code by fetching a string of zeros from the shadow memory, which causes the kernel to terminate the rootkit installation program. Interestingly, when NICKLE operates in rewrite mode, it rewrites the malicious code and forces it to return -1. However, it seems that SucKIT does not bother to check the return value and so the rootkit installation just fails silently and the kernel-level functionality does not work.

In the right-side window in Figure 2, NICKLE reports the authentication and shadowing of sequences of kernel instructions starting from the initial BIOS



```
[root@localhost sk-1.3b]# ./sk
[===== SucKIT version 1.3b, Sep 15 2007 <http://sd
[====== (c)oded by sd <sd@cdi.cz> & devik <devik@c
RK_Init: idt=0xc010022c, sct[]=0xc026b198, kmalloc
Z_Init: Allocating kernel-code memory...Unable to 
eference at virtual address 0000003b
 printing eip:
cfbda044
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<cfbda044>]    Not tainted
EFLAGS: 00000283
eax: 0000003b   ebx: ce784000   ecx: c026b198   ed
esi: c010b390   edi: cfbd8000   ebp: bfffdb18   es
ds: 0018   es: 0018   ss: 0018
Process sk (pid: 726, stackpage=ce785000)
Stack: c0106daf cfbd8000 c026b198 bfffdae0 c010b39
       0000002b 0000002b 0000003b 0804986c 0000002
Call Trace: [<c0106daf>] [<c010b390>]

Code: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

      Got signal 11 while manipulating kernel!
[root@localhost sk-1.3b]# _
```

```
[xjiang@centreville NICKLE]$ ./nickle -hda ../images/redhat8.0.img -no-acpi -m 256
VM physical memory @ [0xa76bf000, 0xb7f0f000); shadow memory @ [0x96e6f000, 0xa76bf000)
Kernel code authenticated and shadowed @ [0x000f0000, 0x00100000) -- BIOS bootstrap
Kernel code authenticated and shadowed @ [0x00100000, 0x00261993) -- kernel text
Kernel code authenticated and shadowed @ [0x00286000, 0x0029b48b) -- kernel init
Kernel module (parport   ) code authenticated and shadowed:
                @ [0x0f24a060, 0x0f24b000) -- parport    (1st page)
                @ [0x0f249000, 0x0f24a000) -- parport    (2nd page)
                @ [0x0f248000, 0x0f249000) -- parport    (3rd page)
                @ [0x0f247000, 0x0f248000) -- parport    (4th page)
Kernel module (parport_pc) code authenticated and shadowed:
                @ [0x0f235060, 0x0f236000) -- parport_pc (1st page)
                @ [0x0f234000, 0x0f235000) -- parport_pc (2nd page)
Kernel module (ieee1394  ) code authenticated and shadowed:
                @ [0x0f293060, 0x0f294000) -- ieee1394   (1st page)
                @ [0x0f292000, 0x0f293000) -- ieee1394   (2nd page)
                @ [0x0f291000, 0x0f292000) -- ieee1394   (3rd page)
                @ [0x0f290000, 0x0f291000) -- ieee1394   (4th page)
                @ [0x0f28f000, 0x0f290000) -- ieee1394   (5th page)
Kernel module (ohci1394  ) code authenticated and shadowed:
                @ [0x0f21f060, 0x0f220000) -- ohci1394   (1st page)
                @ [0x0f21e000, 0x0f21f000) -- ohci1394   (2nd page)
                @ [0x0f21d000, 0x0f21e000) -- ohci1394   (3rd page)
Kernel module (autofs    ) code authenticated and shadowed:
                @ [0x0eecf060, 0x0eed0000) -- autofs     (1st page)
                @ [0x0eece000, 0x0eecf000) -- autofs     (2nd page)
Unauthorized kernel code execution @ 0xcfbda044
                RESPONSE: unauthorized kernel code not executed (Break Mode)
```

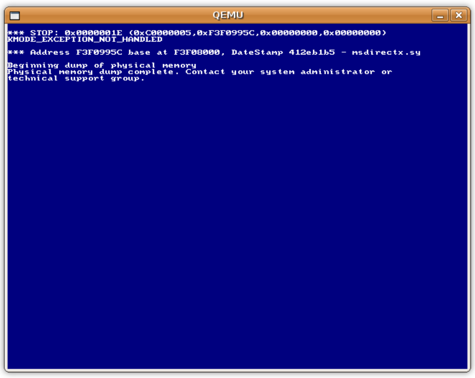
Fig. 2. NICKLE/QEMU+KQEMU foils the SucKIT rootkit (guest OS: RedHat 8.0)

bootstrap code to the kernel text as well as its initialization code and finally to various legitimate kernel modules. In this experiment, there are five legitimate kernel modules, *parport.o*, *parport_pc.o*, *ieee1394.o*, *ohci1394*, and *autofs.o*, all authenticated and shadowed. The code portion of the kernel module begins with an offset of `0x60` bytes in the first page. The first `0x60` bytes are for the kernel module header, which stores pointers to information such as the module's name, size, and other entries linking to the global linked list of loaded kernel modules. This is another example of *mixed kernel pages* with code and data in Linux (Section 2.2).

FU Rootkit Experiment. The FU rootkit [19] is a Windows rootkit that loads a kernel driver and proceeds to manipulate kernel data objects. The manipulation will allow the attacker to hide certain running processes or device drivers loaded in the kernel. When running FU on NICKLE, the driver is unable to load successfully as the driver-specific initialization code is considered unauthorized kernel code. Figure 3 compares NICKLE's two response modes against FU's attempt to load its driver. Under break mode, the OS simply breaks with a blue screen. Under rewrite mode, the FU installation program fails ("Failed to initialize driver.") but the OS does not crash.

4.2 Impact on Performance

To evaluate NICKLE's impact on system performance we have performed benchmark-based measurements on both VMMs – with and without NICKLE. The physical host in our experiments has an Intel 2.40GHz processor and 3GB of RAM running Ubuntu Linux 7.10. QEMU version 0.9.0 with KQEMU 1.3.0pre11 or VirtualBox 1.5.0 OSE is used where appropriate. The VM's guest OS is Redhat 8.0 with a custom compile of a vanilla Linux 2.4.18 kernel and is started inuniprocessor mode with the default amount of memory (256MB for

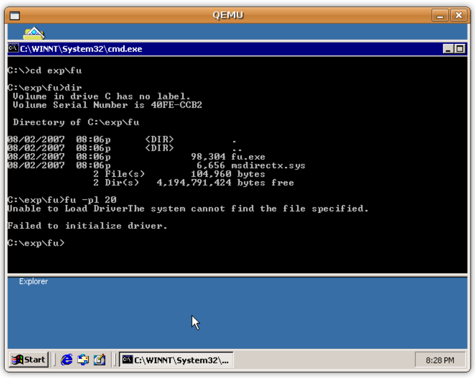


```
*** STOP: 0x0000001E (0xC0000005,0xF3F0995C,0x00000000,0x00000000)
KMODE_EXCEPTION_NOT_HANDLED

*** Address F3F0995C base at F3F08000, DateStamp 412eb1b5 - msdirectx.sy

Beginning dump of physical memory
Physical memory dump complete. Contact your system administrator or
technical support group.
```

(a) Under break mode



```
C:\>cd exp\fu

C:\exp\fu>dir
 Volume in drive C has no label.
 Volume Serial Number is 40FE-CCB2

 Directory of C:\exp\fu

08/02/2007  08:06p      <DIR>          .
08/02/2007  08:06p      <DIR>          ..
08/02/2007  08:06p              98,304 fu.exe
08/02/2007  08:06p               6,656 msdirectx.sys
               2 File(s)        104,960 bytes
               2 Dir(s)   4,194,791,424 bytes free

C:\exp\fu>fu -pl 20
Unable to Load DriverThe system cannot find the file specified.

Failed to initialize driver.

C:\exp\fu>
```

(b) Under rewrite mode

Fig. 3. Comparison of NICKLE/QEMU+KQEMU's response modes against the FU rootkit (guest OS: Windows 2K)

Table 2. Software configuration for performance evaluation

Item	Version	Configuration	Item	Version	Configuration
Redhat	8.0	Using Linux 2.4.18	Apache	2.0.59	Using the default high-performance configuration file
Kernel	2.4.18	Standard kernel compilation	ApacheBench	2.0.40-dev	`-c3 -t 60 <url/file>`
			Unixbench	4.1.0	`-10 index`

Table 3. Application benchmark results

	QEMU+KQEMU			VirtualBox		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Kernel Compiling	231.490s	233.529s	0.87%	156.482s	168.377s	7.06%
`insmod`	0.088s	0.095s	7.34%	0.035s	0.050s	30.00%
Apache	351.714 req/s	349.417 req/s	0.65%	463.140 req/s	375.024 req/s	19.03%

VirtualBox and 128MB for QEMU+KQEMU). Table 2 shows the software configuration for the measurement. For the Apache benchmark, a separate machine connected to the host via a dedicated gigabit switch is used to launch ApacheBench. When applicable, benchmarks are run 10 times and the results are averaged.

Three application-level benchmarks (Table 3) and one micro-benchmark (Table 4) are used to evaluate the system. The first application benchmark is a kernel compilation test: A copy of the Linux 2.4.18 kernel is uncompressed, configured, and compiled. The total time for these operations is recorded and a lower number is better. Second, the `insmod` benchmark measures the amount of time taken to insert a module (in this case, the *ieee1394* module) into the kernel and again lower is better. Third, the ApacheBench program is used to measure the VM's throughput when serving requests for a 16KB file. In this case, higher is better. Finally, the UnixBench micro-benchmark is executed to evaluate the more fine-grained performance impact of NICKLE. The numbers

Table 4. UnixBench results (for the first two data columns, higher is better)

	QEMU+KQEMU			**VirtualBox**		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Dhrystone	659.3	660.0	-0.11%	1843.1	1768.6	4.04%
Whetstone	256.0	256.0	0.00%	605.8	543.0	10.37%
Execl	126.0	127.3	-1.03%	205.4	178.2	13.24%
File copy 256B	45.5	46	-1.10%	2511.8	2415.7	3.83%
File copy 1kB	67.6	68.2	-0.89%	4837.5	4646.9	3.94%
File copy 4kB	128.4	127.4	0.78%	7249.9	7134.3	1.59%
Pipe throughput	41.7	40.7	2.40%	4646.9	4590.9	1.21%
Process creation	124.7	118.2	5.21%	92.1	85.3	7.38%
Shell scripts (8)	198.3	196.7	0.81%	259.2	239.8	7.48%
System call	20.9	20.1	3.83%	2193.3	2179.9	0.61%
Overall	106.1	105.0	**1.01%**	1172.6	1108.7	**5.45%**

reported in Table 4 are an index where higher is better. It should be noted that the benchmarks are meant primarily to compare a NICKLE-enhanced VMM with the corresponding unmodified VMM. These numbers are not meant to compare different VMMs (such as QEMU+KQEMU vs. VirtualBox).

QEMU+KQEMU. The QEMU+KQEMU implementation of NICKLE exhibits very low overhead in most tests. In fact, a few of the benchmark tests show a slight performance gain for the NICKLE implementation, but we consider these results to signify that there is no noticeable slowdown due to NICKLE for that test. From Table 3 it can be seen that both the kernel compilation and Apache tests come in below 1% overheard. The `insmod` test has a modest overhead, 7.3%, primarily due to the fact that NICKLE must calculate and verify the hash of the module prior to copying it into the shadow memory. Given how infrequently kernel module insertion occurs in a running system, this overhead is not a concern. The UnixBench tests in Table 4 further testify to the efficiency of the NICKLE implementation in QEMU+KQEMU, with the worst-case overhead of any test being 5.21% and the overall overhead being 1.01%. The low overhead of NICKLE is due to the fact that NICKLE's modifications to the QEMU control flow only take effect while executing kernel code (user-level code is executed by the unmodified KQEMU accelerator).

VirtualBox. The VirtualBox implementation has a more noticeable overhead than the QEMU+KQEMU implementation, but still runs below 10% for the majority of the tests. The kernel compilation test, for example, exhibits about 7% overheard; while the UnixBench suite shows a little less than 6% overall. The Apache test is the worst performer, showing a 19.03% slowdown. This can be attributed to the heavy number of user/kernel mode switches that occur while serving web requests. It is during the mode switches that the VirtualBox implementation does its work to ensure only verified code will be executed directly [16], hence incurring overhead. The `insmod` test shows a large performance degradation, coming in at 30.0%. This is due to the fact that module insertion on the VirtualBox implementation entails the VMM leaving native code execution as well as verifying the module. However, this is not a concern as module insertion is an uncommon event at runtime. Table 4 shows that the worst performing UnixBench test (Execl) results in an overhead of 13.24%. This result is most likely due to a larger number of user/kernel mode switches that occur during that test.

In summary, our benchmark experiments show that NICKLE incurs minimal to moderate impact on system performance, relative to that of the respective original VMMs.

5 Discussion

In this section, we discuss several issues related to NICKLE. First, the goal of NICKLE is to prevent unauthorized code from executing in the kernel space, but not to protect the integrity of kernel-level control flows. This means that it is possible for an attacker to launch a "return-into-libc" style attack within the kernel by leveraging only the existing authenticated kernel code. Recent work by Shacham [20] builds a powerful attacker who can execute virtually arbitrary code using only a carefully crafted stack that causes jumps and calls into existing code. Fortunately, this approach cannot produce *persistent* code to be called on demand from other portions of the kernel. And Petroni et al. [3] found that 96% of the rootkits they surveyed require persistent code changes. From another perspective, an attacker may also be able to directly or indirectly influence the kernel-level control flow by manipulating certain non-control data [21]. However, without its own kernel code, this type of attack tends to have limited functionality. For example, all four stealth rootkit attacks described in [22] need to execute their own code in the kernel space and hence will be defeated by NICKLE. Meanwhile, solutions exist for protecting control flow integrity [3, 23, 24] and data flow integrity [25], which can be leveraged and extended to complement NICKLE.

Second, the current NICKLE implementation does not support self-modifying kernel code. This limitation can be removed by intercepting the self-modifying behavior (e.g., based on the translation cache invalidation resulting from the self-modification) and re-authenticating and shadowing the kernel code after the modification.

Third, NICKLE currently does not support kernel page swapping. Linux does not swap out kernel pages, but Windows does have this capability. To support kernel page swapping in NICKLE, it would require implementing the introspection of swap-out and swap-in events and ensuring that the page being swapped in has the same hash as when it was swapped out. Otherwise an attacker could modify swapped out code pages without NICKLE noticing. This limitation has not yet created any problem in our experiments, where we did not encounter any kernel level page swapping.

Fourth, targeting kernel-level rootkits, NICKLE is ineffective against user-level rootkits. However, NICKLE significantly elevates the trustworthiness of the guest OS, on top of which anti-malware systems can be deployed to defend against user-level rootkits more effectively.

Fifth, the deployment of NICKLE increases the memory footprint for the protected VM. In the worst case, memory shadowing will double the physical memory usage. As our future work, we can explore the use of demand-paging to effectively reduce the extra memory requirement to the actual amount of memory needed. Overall, it is reasonable and practical to trade memory space for elevated OS kernel security.

Finally, we point out that NICKLE assumes a trusted VMM to achieve the "NICKLE" property. This assumption is needed because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. We also acknowledge that a VM environment can potentially be fingerprinted and detected [26, 27] by attackers so that their malware can exhibit different behavior [28]. We can improve the fidelity of the VM environment (e.g., [29, 30]) to thwart some of the VM detection methods. Meanwhile, as virtualization continues to gain popularity, the concern over VM detection may become less significant as attackers' incentive and motivation to target VMs increases.

6 Related Work

Rootkit Prevention Through Kernel Integrity Enforcement. The first area of related work includes recent efforts in enforcing kernel integrity to thwart kernel rootkit installation or execution. Livewire [6], based on a software-based VMM, aims at protecting the guest OS kernel code and critical data structures from being modified. However, an attacker may choose to load malicious rootkit code into the kernel space without manipulating the original kernel code.

SecVisor [7] is a closely related work that leverages new hardware extensions to enforce life-time kernel integrity and provide a guarantee similar to "NICKLE". However, there are two main differences between SecVisor and NICKLE: First, the deployment of SecVisor requires modification to OS kernel source code as well as the latest hardware support for MMU and IOMMU virtualization. In comparison, NICKLE is a guest-transparent solution that supports guest OSes "as is" on top of legacy hardware platforms. In particular, NICKLE does not rely on the protection of any guest OS data structures (e.g., the GDT – global descriptor table). Second, SecVisor is developed to enforce the W$\oplus$X principle for the protected VM kernel code. This principle intrinsically conflicts with mixed kernel pages, which exist in current OSes (e.g., Linux and Windows). NICKLE works in the presence of mixed kernel pages. OverShadow [31] adopts a similar technique of memory shadowing at the VMM level with the goal of protecting application memory pages from modification by even the OS itself. In comparison, NICKLE has a different goal and aims at protecting the OS from kernel rootkits.

To ensure kernel code integrity, techniques such as driver signing [32] as well as various forms of driver verification [5, 33] have also been proposed. These techniques are helpful in verifying the identity or integrity of the loaded driver. However, a kernel-level vulnerability could potentially be exploited to bypass these techniques. In comparison, NICKLE operates at the lower VMM level and is capable of blocking zero-day kernel-level exploitations.

Symptom-Driven Kernel Rootkit Detection. The second area of related work is the modeling and specification of symptoms of a rootkit-infected OS kernel which can be used to detect kernel rootkits. Petroni et al. [4] and Zhang et al. [34] propose the use of external hardware to grab the runtime OS memory image and detect possible rootkit presence by spotting certain kernel code integrity violations (e.g., rootkit-inflicted kernel code manipulation). More recent works further identify possible violations of semantic integrity of dynamic kernel data [2] or state based control-flow integrity of kernel code [3]. Generalized control-flow integrity [23] may have strong potential to be used as a prevention technique, but as yet has not been applied to kernel integrity. Other solutions such as Strider GhostBuster [35] and VMwatcher [1] target the self-hiding nature of rootkits and infer rootkit presence by detecting discrepancies between the views of the same system from different perspectives. All the above approaches are, by design, for the *detection* of a kernel rootkit *after* it has infected a system. Instead, NICKLE is for the *prevention* of kernel rootkit execution in the first place.

Attestation-Based Rootkit Detection. The third area of related work is the use of attestation techniques to verify the software running on a target platform. Terra [13] and other code attestation schemes [36, 37, 38] are proposed to verify software that is being located into the memory for execution. These schemes are highly effective in providing the *load-time* attestation guarantee. Unfortunately, they are not able to provide *run-time* kernel integrity.

7 Conclusion

We have presented the design, implementation, and evaluation of NICKLE, a VMM-based approach that transparently detects and prevents the launching of kernel rootkit attacks against guest VMs. NICKLE achieves the "NICKLE" guarantee, which foils the common need of existing kernel rootkits to execute their own unauthorized code in the kernel space. NICKLE is enabled by the scheme of memory shadowing, which achieves guest transparency through the guest memory access indirection technique. NICKLE's portability has been demonstrated by its implementation in three VMM platforms. Our experiments show that NICKLE is effective in preventing 23 representative real-world kernel rootkits that target a variety of commodity OSes. Our measurement results show that NICKLE adds only modest overhead to the VMM platform.

Acknowledgements. The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by NSF Grants CNS-0716376, CNS-0716444 and CNS-0546173.

References

[1] Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)

[2] Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the 15th USENIX Security Symposium (2006)

[3] Petroni Jr., N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)

[4] Petroni, N., Fraser, T., Molina, J., Arbaugh, W.: Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings of the 13th USENIX Security Symposium, pp. 179–194 (2004)

[5] Wilhelm, J., Chiueh, T.-c.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)

[6] Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. Network and Distributed Systems Security Symposium (NDSS 2003) (February 2003)

[7] Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 2007) (October 2007)

[8] Bellard, F.: QEMU: A Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)

[9] Innotek: Virtualbox (Last accessed, September 2007), `http://www.virtualbox.org/`

[10] Intel: Vanderpool Technology (2005), `http://www.intel.com/technology/computing/vptech`

[11] AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.12 edition (September 2006)

[12] Dunlap, G., King, S., Cinar, S., Basrai, M., Chen, P.: ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In: Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002) (2002)

[13] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: Proc. of ACM Symposium on Operating System Principles (SOSP 2003) (October 2003)

[14] Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)

[15] Joshi, A., King, S., Dunlap, G., Chen, P.: Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In: Proc. ACM Symposium on Operating Systems Principles (SOSP 2005), pp. 91–104 (2005)

[16] Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. Technical report CERIAS TR 2001-146, Purdue University

[17] Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: Proceedings of IEEE Symposium on Security and Privacy, May 1997, pp. 65–71 (1997)

[18] sd, devik: Linux on-the-fly Kernel Patching without LKM. Phrack 11(58) Article 7

[19] fuzen_op: Fu rootkit (Last accessed, September 2007), `http://www.rootkit.com/project.php?id=12`

[20] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)

[21] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.: Non-Control-Data Attacks Are Realistic Threats. In: Proceedings of the 14th USENIX Security Symposium (August 2005)

[22] Baliga, A., Kamat, P., Iftode, L.: Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In: Proc. of IEEE Symposium on Security and Privacy (Oakland 2007) (May 2007)

[23] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control Flow Integrity: Principles, Implementations, and Applications. In: Proc. ACM Conference on Computer and Communications Security (CCS 2005) (November 2005)

[24] Grizzard, J.B.: Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems. Ph.D. Thesis, Georgia Institute of Technology (May 2006)

[25] Castro, M., Costa, M., Harris, T.: Securing Software by Enforcing Data-Flow Integrity. In: Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006) (2006)

[26] Klein, T.: Scooby Doo - VMware Fingerprint Suite (2003), http://www.trapkit.de/research/vmm/scoopydoo/index.html

[27] Rutkowska, J.: Red Pill: Detect VMM Using (Almost) One CPU Instruction (November 2004), http://invisiblethings.org/papers/redpill.html

[28] F-Secure Corporation: Agobot, http://www.f-secure.com/v-descs/agobot.shtml

[29] Kortchinsky, K.: Honeypots: Counter Measures to VMware Fingerprinting (January 2004), http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html

[30] Liston, T., Skoudis, E.: On the Cutting Edge: Thwarting Virtual Machine Detection (2006), http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

[31] Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In: Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008) (March 2008)

[32] Microsoft Corporation: Driver Signing for Windows, http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/code_signing.mspx?mfr=true

[33] Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Yew, P.-C., Xue, J. (eds.) ACSAC 2004. LNCS, vol. 3189, pp. 91–100. Springer, Heidelberg (2004)

[34] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., Sailer, R.: Secure Coprocessor-based Intrusion Detection. In: Proceedings of the 10th ACM SIGOPS European Workshop, pp. 239–242 (2002)

[35] Wang, Y.M., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. In: Proc. IEEE International Conference on Dependable Systems and Networks (DSN 2005), pp. 368–377 (2005)

[36] Kennell, R., Jamieson, L.H.: Establishing the Genuinity of Remote Computer Systems. In: Proc. of the 12th USENIX Security Symposium (August 2003)

[37] Sailer, R., Jaeger, T., Zhang, X., van Doorn, L.: Attestation-based Policy Enforcement for Remote Access. In: Proc. of ACM Conference on Computer and Communications Security (CCS 2004) (October 2004)

[38] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: Proc. of the 13th USENIX Security Symposium (August 2004)