

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

Ryan Riley
Purdue University
rileyrd@cs.purdue.edu

Xuxian Jiang
George Mason University
xjiang@gmu.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.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, VMware Workstation, and VirtualBox. 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 (e.g., < 2% to QEMU+KQEMU).

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. Unfortunately, a large body of the existing anti-malware techniques relies on the trustworthiness of the OS kernel, making them intrinsically ineffective against kernel rootkits.

There have been a number of recent efforts in mitigating the threat of kernel rootkits and they can be classified into two main categories: (1) detecting the presence of kernel rootkits in a system [3, 23, 25, 32, 33, 42, 47] and (2) preventing the compromise of OS kernel integrity [19, 41]. In the first category, Copilot [33] 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 [25] further extends that capability by detecting the violation of kernel integrity using semantic specifications of static and dynamic kernel data. SBCFI [32] detects violations of the kernel’s control flow integrity using the kernel’s control-flow graph. Other solutions [3, 23, 45] compare a kernel’s spatial or temporal semantic views to detect possible view discrepancies caused by rootkit infections. One common attribute of all the

above approaches 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 [19], 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) and these malicious instructions can still enable backdoor access and/or privilege escalation for the attacker. More recently, SecVisor [41] is proposed as an elegant 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 2.2.1).

To address the limitations of 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 capable of *transparently* protecting unmodified guest OSes (e.g., Fedora Core 3/4/5 and Windows 2K/XP) against kernel rootkits on a legacy hardware platform. 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, 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 $W\oplus X$ 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 (Section 2.2.1), this approach is not viable for guest-transparent protection. To address that, we propose a VMM-based *memory shadowing* scheme for NICKLE. 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. Moreover, memory shadowing works in the face of mixed kernel pages.

We have implemented NICKLE in three different VMMs: QEMU[13] (with the KQEMU [4] acceler-

¹With 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.

ator), VMware Workstation [8], and VirtualBox [7]. 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.

The rest of the paper is organized as follows: Section 2 presents the goals and enabling techniques of NICKLE. Section 3 describes NICKLE implementation details. Evaluation results are presented in Section 4. Several issues related to NICKLE are discussed in Section 5. Section 6 discusses related work. Finally, Section 7 concludes this paper.

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 2.2.1). We note that the goal and 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. To address this challenge, we develop the efficient technique of guest memory access indirection (Section 2.2.2).

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 [9, 15] 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. As such, the kernel rootkit can compromise arbitrary entities and facilities inside the VM – including the guest OS kernel – but it cannot

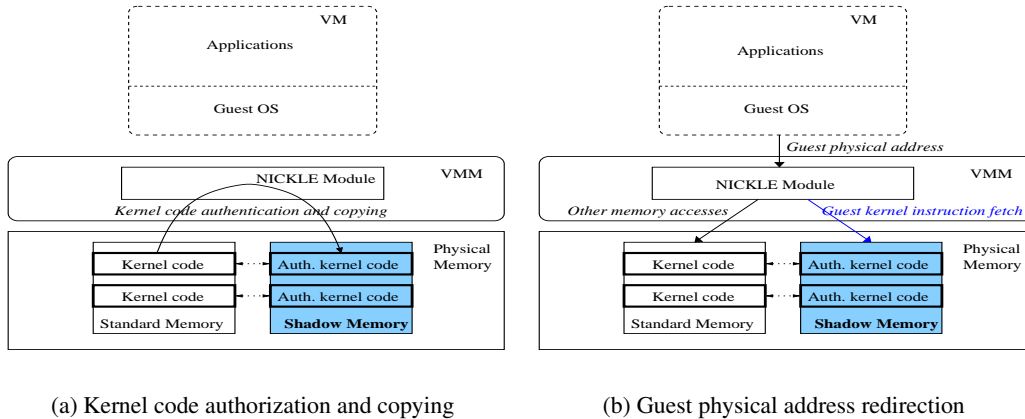


Figure 1: Memory shadowing scheme in NICKLE

break out of the VM and corrupt the underlying VMM. This model is based on the observation that the code base of a VMM is relatively small and stable compared with that of a legacy OS. Further, the VMM provides a rather limited interface (which can be further hardened) to the VMs. This assumption is shared by many other VMM-based security research efforts [16, 18, 19, 22, 23, 24]. 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

2.2.1 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). However, the shadow memory is transparent to the VM and is controlled by the VMM instead. 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), details in Section 2.2.2). 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 after verifying the instruction by comparing the shadow and standard memories (Figure 1(b), details in Section 2.2.2). 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, which stores only 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 $W\oplus X$?” In essence, $W\oplus X$ 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

²There 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.

from, code injection becomes impossible. There are two main reasons why this scheme is not adequate for stopping kernel level rootkits:

First, $W \oplus X$ 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 memory³, which means that the first two kernel pages cover memory ranges $[0xc0000000, 0xc0200000)$ 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 $W \oplus X$ 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. Detailed page table dumps showing mixed kernel pages in both Windows XP and Linux kernels can be found at <http://www.ise.gmu.edu/~xjiang/NICKLE>. On the other hand, NICKLE enforces restricted access/execution at the (finer) granularity of individual kernel instructions and thus is not affected by the mixed page existence.⁴

Second, $W \oplus X$ 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 $W \oplus X$, but $W \oplus KX$ (i.e. not writable and kernel-executable at the same time.) Still, we point out that the enforcement of $W \oplus KX$ 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. In fact, SecVisor [41] strives to implement such permission control by devising an approach to ensure that all user/kernel mode transitions are captured and permissions changed accordingly.

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 and enforce execution/writing accordingly?” 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 $O(\log N)$ search time. The search will be performed upon *every* kernel instruction fetch thus incurring high computation overhead. 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 for each kernel instruction execution. In addition, memory shadowing makes it convenient to compare the original authenticated kernel instruction in the shadow memory and the “mirror” (and possibly modified) instruction in the standard memory. If they differ (indicating malicious kernel code injection), a number of response actions can be implemented based on the difference (details in Section 3).

2.2.2 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

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

⁴We 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 (see the above URL for detailed page table dump). NICKLE instead simply avoids such complexity and works even with mixed kernel pages.

code in the kernel space? We note that our solutions have to be transparent to the guest OS (and thus to the kernel rootkits). In this section, we present the guest memory access indirection technique to address these issues.

Guest memory access indirection is performed between the VM and its memory (standard and shadow) by a thin NICKLE module inside the VMM (Figure 1). It has two main functions: (1) kernel code authentication and copying at VM startup and upon kernel module loading (Figure 1(a)) and (2) guest physical address redirection at runtime (Figure 1(b)).

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 NICKLE needs to be invoked to verify kernel code and shadow the authorized code: 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 and the system bootstrap code will first be verified and copied into the shadow memory. The bootstrap code then loads and decompresses the Linux kernel into the guest’s standard memory. 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 [11]) 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. LKM support is helpful in that OS distributions can ship with kernels that have a relatively small memory footprint and grow dynamically based on users’ needs. However, from NICKLE’s perspective, LKMs are considered injected kernel code and should 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 [22, 23]. When NICKLE detects the loading of a new kernel module, it intercepts the VM’s execution and performs kernel module code authentication and shadowing. When NICKLE detects the unloading of an existing kernel module, it can remove the corresponding code segment from the shadow memory. 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 maintainer⁵. 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 (e.g., running the kernel module insertion code incorrectly, or using an alternative method to insert a module), *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 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)).

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

We point out that the interception of VM memory accesses can be provided by existing VMMs (e.g., VMware, QEMU+KQEMU, and VirtualBox). 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.

2.2.3 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 three response modes facilitated by the current NICKLE system:

Observe mode: NICKLE will simply log the attempt and allow the malicious kernel code execution to proceed by routing the instruction fetch to the standard memory. NICKLE in this mode is a real-time kernel rootkit monitoring system without prevention actions. This can be useful for systems such as honeypots.

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. Our NICKLE implementation supports this mode for both Windows and Linux.

Break mode: NICKLE will take no action and route the instruction fetch to the *shadow memory*. Different from observe mode, the rootkit’s malicious code will not be executed in this mode. 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. For example, Linux will terminate only the offending process while Windows will cause a complete system halt (this is discussed further in Section 4).

3 NICKLE Implementation

To validate the portability of the NICKLE design, we have implemented NICKLE in three different VMMs: QEMU+KQEMU [13], VMware Workstation [8]⁶, and VirtualBox [7]. 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 implementation in supporting Windows guest OSes.

⁶We are grateful to VMware for providing the source code of VMware Workstation 6.0, which makes the NICKLE port possible.

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 VMware Workstation and 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.

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 then
3     Authenticate and shadow kernel text;
4   else
5     if a kernel module is being loaded then
6       Authenticate and shadow kernel module code;
7     else
8       Unauthorized execution attempt - Execute response;
9     end
10  end
11 else
12   Fetch, translate, and cache code;
13 end
```

In the above algorithm, 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 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. To intercept and interpret VM system calls at the VMM level, we leverage part of our earlier implementation for “out of the box” capture and reconstruction of VM semantic states and events. More details are presented in [22, 23].

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. This capability is necessary for LKM support but unfortunately introduces difficulty to NICKLE, as 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 will show a running example in Section 4 where kernel bootstrapping and kernel module loading events are transparently intercepted and validated by NICKLE.

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 in 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 – with flexible responses (Section 3.2) – a capability not provided by the existing approaches [19, 41]. 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 a variety of 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 that initiated the unauthorized kernel code execution. This outcome allows Linux to run unhindered after the response. 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. For example, for a Windows XP guest OS, NICKLE can rewrite the instructions with “safe” instructions so that Windows can still proceed instead of halting with a blue screen. 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.

3.3 Porting Experience

We have experienced no major difficulty in porting NICKLE to other VMMs. The NICKLE implementations in all three VMMs are lightweight: The SLOC [46] (source lines of code) added to implement NICKLE

in QEMU+KQEMU, VMware Workstation, and VirtualBox are 853, 1181, and 762, 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 and VMware Workstation 6.0.

Further VirtualBox Optimizations The VirtualBox VMM is a software VMM, but attempts to execute as much guest code (both user and kernel) as possible directly on the host processor in user mode. In the event that a piece of guest kernel code cannot be executed directly, VirtualBox makes use of parts of the QEMU source code (namely the recompiler) to do binary translation. Our original port simply reused the NICKLE code for QEMU+KQEMU and modified VirtualBox to execute all kernel code using the QEMU recompiler. As one would expect, this caused significant performance degradation due to the speed difference between native and recompiler-based executions.

To achieve better performance, we use the following optimization: If a kernel page contains nothing but verified kernel code, then the code from the page will be executed directly on the host processor; For a kernel page mixed with both code and data, the execution will be passed off to the recompiler and the related memory requests will be mediated. It turns out that this technique can result in significant performance gains: in the kernel compilation test (Section 4), NICKLE before optimization incurred a 50% slowdown while after optimization it is reduced to 7.06%. Finally, we point out that the VirtualBox port is, by far, the most difficult due to the complexity of the VMM itself. As such we still consider this port to be a proof of concept and reasonable indicator of performance, but additional time would be required to further reduce the performance hit and make it comparable to other VMM ports.

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 rootkits⁷ 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 our experimental results: NICKLE is able to detect and prevent the execution of malicious kernel code in *all* experiments. Furthermore, we evaluate NICKLE’s three response modes (observe, rewrite, and break) against each rootkit. The outcomes under these response modes are also reported in Table 1. Finally, we note that NICKLE in all three VMMs is able to achieve the same results. In the following, we present more details about the experiments.

SucKIT Rootkit Experiment The SucKIT rootkit [40] 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. The execution of the string of zeroes leads the kernel to report the error and terminate the rootkit installation program. Interestingly, when NICKLE operates in rewrite mode, NICKLE rewrites the malicious code and forces it to return `-1`. However, it seems that SucKIT does not bother to check the return value. As a result, the rootkit installation just fails silently and SucKIT’s kernel-level functionality will not work.

⁷There 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.

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response					
			Observe Mode		Rewrite Mode		Break Mode	
			Detected?	Prevented?	Prevented?	Outcome	Prevented?	Outcome
Linux 2.4	adore 0.42, 0.53	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	adore-ng 0.56	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	knark	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	rkit 1.01	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	kbdv3	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	allroot	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	rial	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	Phantasmagoria	LKM	✓	✓	in.smod fails	✓	Seg. fault	
SucKIT 1.3b	/dev/kmem	✓	✓	Installation fails silently	✓	Seg. fault		
Linux 2.6	adore-ng 0.56	LKM	✓	✓	in.smod fails	✓	Seg. fault	
	eNYeLKM v1.2	LKM	✓	✓	in.smod 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	✓	✓	in.smod 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	
	yyLhac	Driver	✓	✓	Driver loading fails	✓	BSOD	
	NT Rootkit	Driver	✓	✓	Driver loading fails	✓	BSOD	

Table 1: Effectiveness of NICKLE in detecting and preventing 23 real-world kernel rootkits ($DKOM^{\dagger}$ is a common rootkit technique which directly manipulates kernel objects; “*partial*”[‡] means the in-kernel component of the Hacker Defender rootkit fails; $BSOD^{\S}$ stands for “Blue Screen Of Death”)

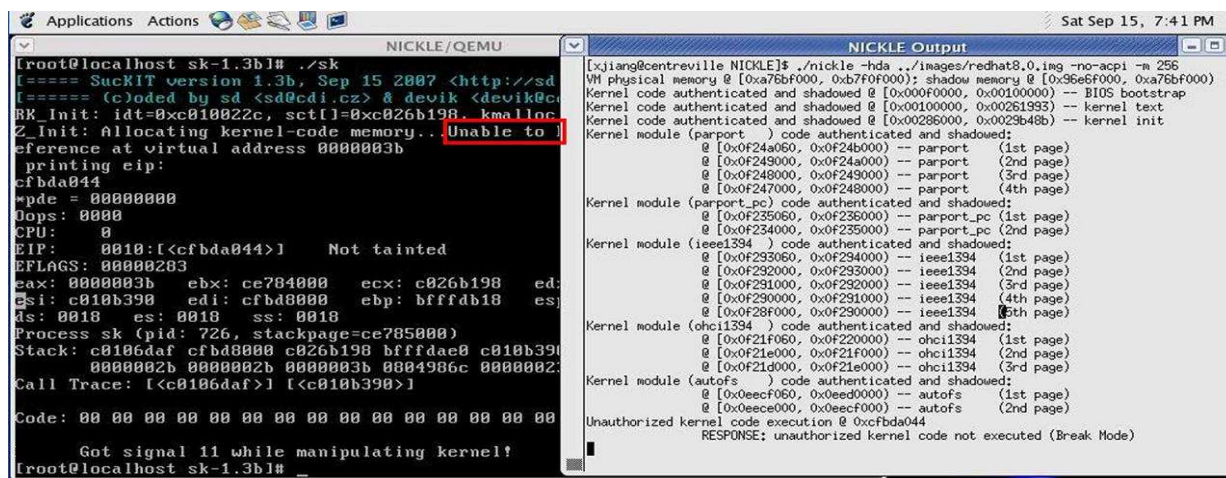
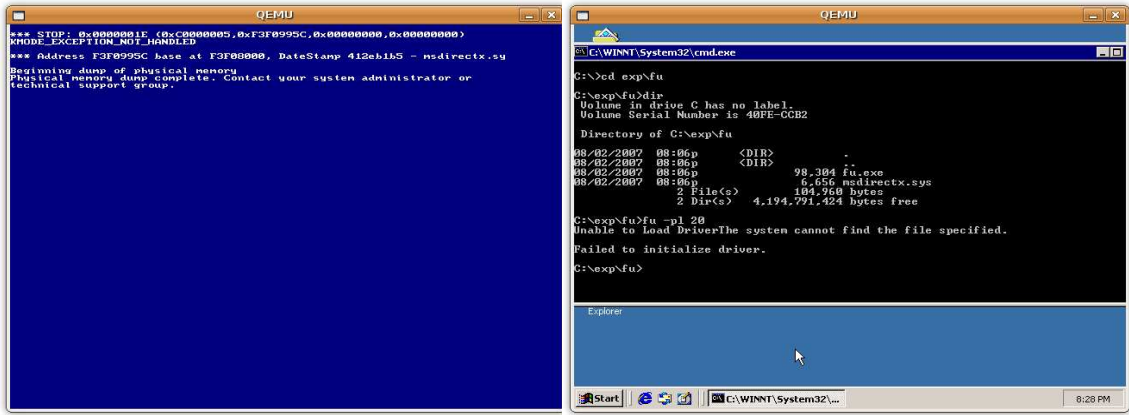


Figure 2: NICKLE/QEMU+KQEMU foils the SucKIT rootkit (guest OS: RedHat 8.0)

In the right-side window in Figure 2, NICKLE reports the authentication and shadowing of sequences of kernel instructions starting from the initial BIOS 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. A keen reader may also notice that the pages of each authenticated kernel module have *descending* physical addresses. This is due to the memory allocation scheme of the Linux kernel which allocates physical pages in descending order. In addition, 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



(a) Under break mode

(b) Under rewrite mode

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

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 2.2.1).

FU Rootkit Experiment The FU rootkit [17] 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 (break mode and rewrite mode) against FU’s attempt to load its driver. Under break mode, the VM simply breaks with a blue screen (Figure 3(a)). Under rewrite mode, the FU installation program fails (“Failed to initialize driver.”) but the VM does not crash (Figure 3(b)).

Due to space constraints, we omit detailed descriptions of other experiments. The Appendix shows screenshots of a number of selected experiments with NICKLE in the three VMMs: Figure A1 shows NICKLE in QEMU+KQEMU preventing the adore-ng rootkit in rewrite mode; Figure A2 compares the break and rewrite modes of NICKLE in VirtualBox against the adore-ng rootkit; Figure A3 shows NICKLE in VMware Workstation preventing the adore-ng-2.6 rootkit in a VM running an off-the-shelf Fedora Core 3 guest OS.

In summary, the kernel rootkit experiments show that NICKLE achieves the “NICKLE” property in a real-time, guest-transparent manner in all three VMMs. We highlight that all the experiments are performed *without* modifying the guest OS kernel code and do not require any rootkit signatures, kernel integrity models or specifications.

4.2 Impact on Performance

To evaluate NICKLE’s impact on system performance we have performed benchmark-based measurements on all three VMMs – with and without NICKLE. We describe details of the QEMU+KQEMU and VirtualBox results, followed by a summary of the VMware tests. 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. For both VMM platforms the VM’s guest OS is Redhat 8.0 with a custom compile of a vanilla Linux 2.4.18 kernel and is started in uniprocessor mode with the default amount of memory (256MB for VirtualBox and 128MB for QEMU+KQEMU). Table 2 shows the software configuration for the measurement. For the Apache benchmark, a separate machine

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

Table 2: Software configuration for performance evaluation

Benchmark	QEMU+KQEMU			VirtualBox		
	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%

Table 3: Application benchmark results

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 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).

Benchmark	QEMU+KQEMU			VirtualBox		
	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%

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

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% overhead. 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), and even in that instance only adds one additional `if` statement for the common case.

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% overhead; 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 (Section 3.3), hence incurring overhead. The `insmod` test shows a large performance degradation, coming in at 30.0%. This is due to the fact that inserting a kernel module on the VirtualBox implementation entails the VMM dropping out of native code execution as well as hashing and verifying the module. However, we point out that, like the QEMU+KQEMU implementation, this overhead is not a concern due to the fact that module insertion is an uncommon event in a running system. Table 4 shows that the worst performing UnixBench test (Exec1) 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.

VMware Workstation The tests on VMware Workstation see a minimal performance impact from NICKLE. However, time constraints prevent us from meeting all of VMware’s license requirements for the publication of detailed performance numbers and we will report those results in an upcoming technical report.

In summary, our benchmark experiments show that NICKLE incurs minimal (QEMU+KQEMU, VMware Workstation) to moderate (VirtualBox) 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 [43] 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. [32] 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 [14]. However, without its own kernel code, this type of attacks tends to have limited functionality. For example, all four stealth rootkit attacks described in [12] 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 [10, 21, 32] and data flow integrity [20], which can be leveraged and extended to complement NICKLE.

Second, the current NICKLE implementation does not support self-modifying kernel code. If the guest OS kernel requires any self-modifying code to run at the kernel level, NICKLE will not allow it to function. However, 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 that 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 will 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. Some parallel efforts in building highly trusted VMMs will be described in Section 6. We also acknowledge that a VM environment can potentially be fingerprinted and detected [28, 35] by attackers. In fact, a number of recent malware systems are able to check whether they are running inside a VM and if so, exhibit different behavior [1]. We can improve the fidelity of the VM environment (e.g., [29, 31]) 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 [19], 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 [41] is a closely related work that leverages new hardware extensions to enforce life-time kernel integrity and provides 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) to enforce its protection. Second, SecVisor is developed to enforce the $W \oplus X$ principle for the protected VM kernel code. The $W \oplus X$ principle intrinsically conflicts with mixed kernel pages, which exist in current OSes (e.g., Linux and Windows). NICKLE avoids such conflict and works in the presence of mixed kernel pages.

To ensure kernel code integrity, techniques such as driver signing [2] have been proposed requiring each kernel driver to be digitally signed by a trusted entity and the kernel loading process will verify the digital signature and refuse to load unsigned drivers. Similarly, Limbo [47] checks the legitimacy of every kernel driver based on its binary content and run-time behavior before loading it into the kernel. Kruegel et al. [30] proposes a binary analysis approach to determine, at load time, if a module’s behavior resembles the behavior of a rootkit. These techniques are helpful in verifying the identity or integrity of the loaded driver. However, being implemented and enforced at the (vulnerable) kernel or user level implies that any 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 by detecting and foiling any attempt to execute unauthorized exploitation code.

Symptom-driven Kernel Rootkit Detection The second area of related work is the modeling and spec-

ification of symptoms of a rootkit-infected OS kernel so that the model and specification can be used to detect kernel rootkits. Petroni et al. [33] and Zhang et al. [48] propose the use of a secure external hardware device 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 [25] or control-flow integrity of kernel code [32]. Other solutions such as Strider GhostBuster [45], RootkitRevealer [5], F-Secure’s Blacklight [3], IceSword [34], and VMwatcher [23] 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 [18] and other code attestation schemes [26, 38, 39] are proposed to verify software that is being loaded 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. Pioneer [42] and BIND [44] aim at verifying the software stack running on the attestation platform. For an ongoing rootkit attack they are able to accurately report runtime verification failures; however, they are not able to prevent the kernel rootkit attack from taking place.

Emerging Rootkit Threats Recently, the threat of emerging virtualization-based rootkits has been demonstrated [27, 36, 49]. King et al. [27] proposes the VM-based rootkit (VMBR) which can be dynamically inserted under an existing OS. Rutkowska et al. [36] further implements a hardware virtualization-based rootkit prototype called “Blue Pill”, claiming the creation of 100% undetectable malware. The Vitriol [49] rootkit, another hardware virtualization-based rootkit, independently confirms this significant threat. These emerging threats can be mitigated or even defeated by recent work on secure booting [11] and secure VMMs (e.g., sHype [37] and TRANGO [6]). With secure booting, VMMs will maintain the lowest-level access to the system thus preventing them from being subverted. Paralleling these efforts, NICKLE assumes the non-subvertability of VMMs in anticipation of wide deployment of anti-subversion solutions.

7 Conclusions

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 small overhead to the VMM platform.

References

- [1] Agobot. <http://www.f-secure.com/v-descs/agobot.shtml>.
- [2] Driver Signing for Windows. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/code_signing.mspx?mfr=true.
- [3] F-Secure Blacklight. <http://www.f-secure.com/blacklight/blacklight.html>. Last accessed August 2007.

- [4] QEMU Accelerator User Documentation. <http://fabrice.bellard.free.fr/qemu/kgemu-doc.html>. Last accessed August 2007.
- [5] RootkitRevealer. <http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.msp>.
- [6] TRANGO, the Real-Time Embedded Hypervisor. <http://www.trango-systems.com/>.
- [7] VirtualBox. <http://www.virtualbox.org/>. Last accessed September 2007.
- [8] VMware Workstation. <http://www.vmware.com/products/ws/>. Last accessed September 2007.
- [9] Vanderpool Technology. <http://www.intel.com/technology/computing/vptech>, 2005.
- [10] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM CCS, Alexandria, VA*, November 2005.
- [11] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.
- [12] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. *Proc. of the 2007 IEEE Symposium on Security and Privacy*, May 2007.
- [13] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.
- [14] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravi Iyer. Non-control-data Attacks are Realistic Threats. In *Proc. USENIX Security Symposium*, August 2005.
- [15] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.12 edition. September 2006.
- [16] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36:211–224, 2002.
- [17] fuzen_op. Fu Rootkit. <http://www.rootkit.com/project.php?id=12>. Last accessed September 2007.
- [18] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, October 2003.
- [19] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.
- [20] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proc. USENIX Security Symposium*, August 2006.
- [21] Julian B. Grizzard. Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems. Ph.D. thesis, Georgia Institute of Technology, May 2006.
- [22] Xuxian Jiang and Xinyuan Wang. “Out-of-the-Box” Monitoring of VM-based High-Interaction Honey-pots. In *Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.
- [23] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 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)*, October 2007.
- [24] Ashlesha Joshi, Samuel T. King, George W. Dunlap, Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.
- [25] Nick L. Petroni., Timothy Fraser, Aaron Walters, and William A. Arbaugh. An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

- [26] Rick Kennel and Leah H Jamieson. Establishing the Genuity of Remote Computer Systems. In *Proc. of the 12th conference on USENIX Security Symposium*, August 2003.
- [27] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing Malware with Virtual Machines. *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [28] Tobias Klein. Scooby Doo - VMware Fingerprint Suite. <http://www.trapkit.de/research/vmm/scoopydoo/index.html>, 2003.
- [29] Kostya Kortchinsky. Honeypots: Countermeasures to VMware Fingerprinting. <http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html>, January 2004.
- [30] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2004.
- [31] Tom Liston and Ed Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, 2006.
- [32] Nick L. Petroni, and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2007.
- [33] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [34] PJF. Icesword. <http://www.antirootkit.com/software/IceSword.htm>, <http://pjf.blogcn.com/>. Last accessed August 2007.
- [35] Joanna Rutkowska. Red Pill: Detect VMM using (almost) One CPU Instruction. <http://invisiblethings.org/papers/redpill.html>, November 2004.
- [36] Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. *Blackhat 2006*, August 2006.
- [37] Reiner Sailer, Enrique Valdez, Trent Jaeger, Ronald Perez, Leendert van Doorn, John Linwood Griffin, and Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. *IBM Research Report RC23511*, February 2005.
- [38] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based Policy Enforcement for Remote Access. In *Proc. of the 11th ACM conference on Computer and communications security*, October 2004.
- [39] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proc. of the 13th conference on USENIX Security Symposium*, August 2004.
- [40] sd and devik. Linux On-the-Fly Kernel Patching without LKM. *Phrack*, 11(58). Article 7.
- [41] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.
- [42] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.
- [43] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of CCS 2007*, October 2007.
- [44] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In *Proc. of the 2005 IEEE Symposium on Security and Privacy*, May 2005.
- [45] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of DSN 2005*, June 2005.

- [46] David A Wheeler. Counting Source Lines of Code (SLOC). <http://www.dwheeler.com/sloc/>. Last accessed September 2007.
- [47] Jeffrey Wilhelm and Tzi-cker Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.
- [48] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.
- [49] Dino Dai Zovi. Hardware Virtualization Based Rootkits. *Blackhat 2006*, August 2006.

Appendix

A Screenshots of Selected Rootkit Experiments on Three VMMs

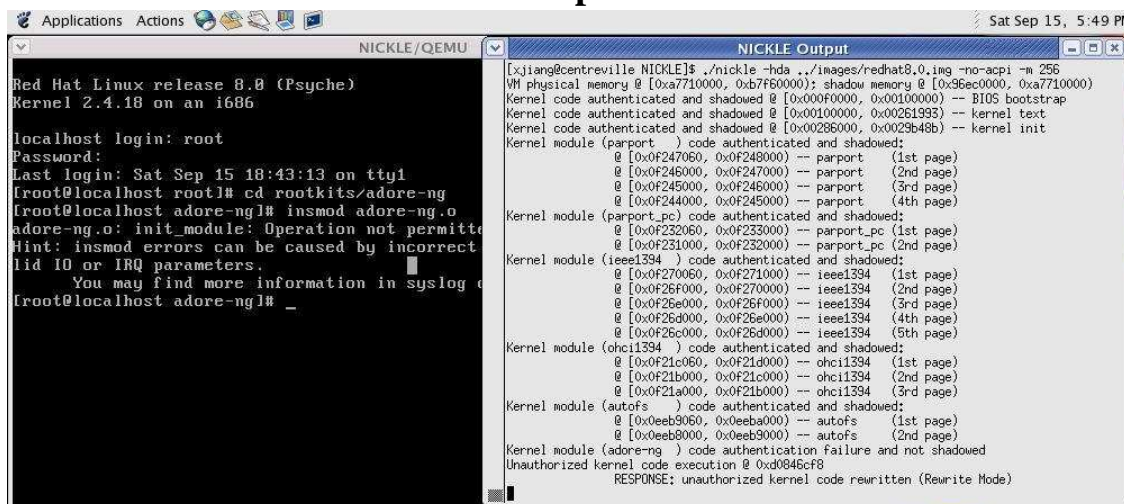
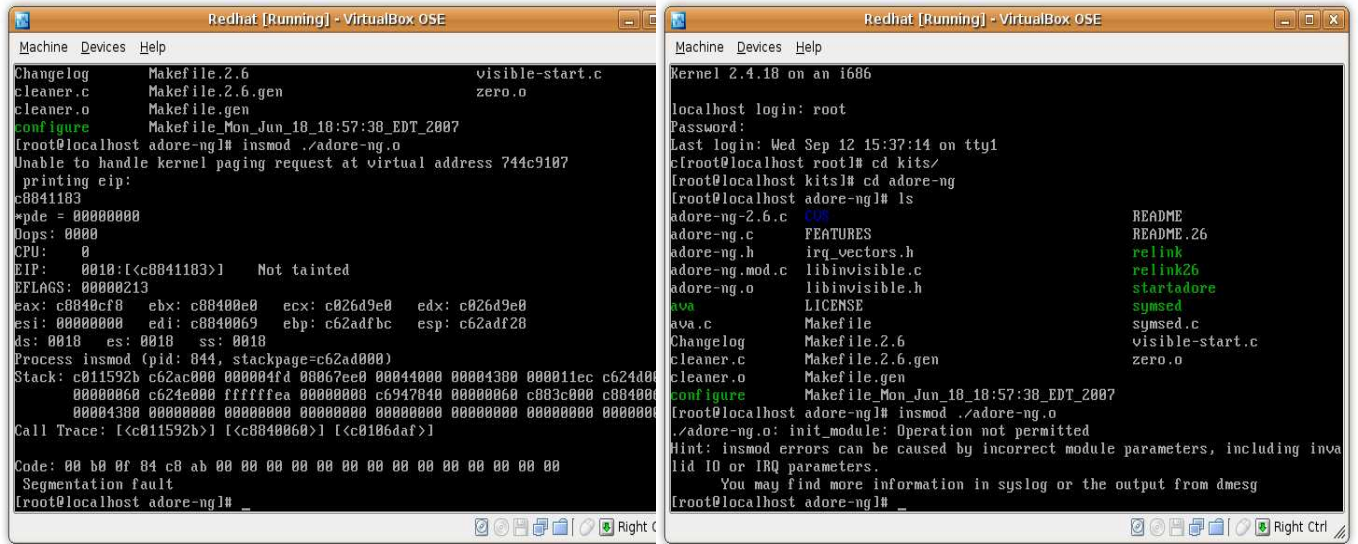


Figure A1: NICKLE/QEMU+KQEMU defeats the adore-ng rootkit (guest OS: RedHat 8.0). As mentioned in Table 1, the adore-ng Linux 2.4/2.6 rootkit is a loadable kernel module that infects and modifies the running OS kernel. Under normal circumstances the module is compiled and inserted into the kernel using the `insmod` command. `insmod` allocates memory space for the module’s code, injects the code into the kernel’s address space, and then attempts to call the module’s `init()` function. With NICKLE (in rewrite response mode), before any instruction in the `init()` is executed, NICKLE rewrites the code to return `-1`. This causes `insmod` to believe that there has been an error during module initialization so it unloads the module automatically. The figure also shows the authentication and shadowing of good kernel code and kernel modules before the rootkit attack.



(a) Under break mode

(b) Under rewrite mode

Figure A2: NICKLE/VirtualBox defeats the adore rootkit (guest OS: RedHat 8.0). The two sub-figures compare NICKLE’s response under break mode (“segmentation fault”) and under rewrite mode (“insmod failure”).

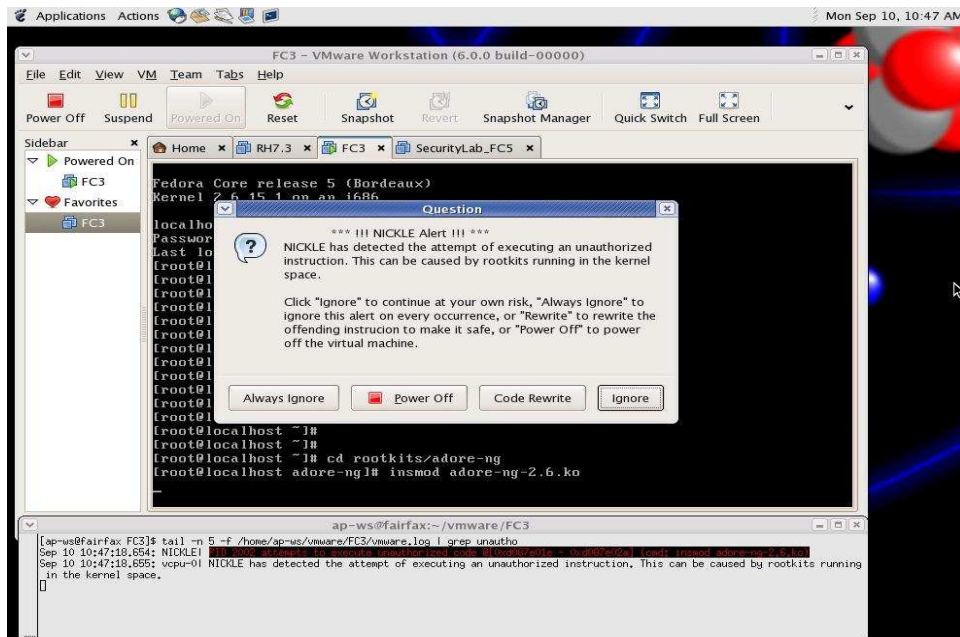


Figure A3: NICKLE/VMware defeats the adore-ng-2.6 rootkit (guest OS: Fedora Core 3). The user is given a number of response options.