Countering Kernel Rootkits with Lightweight Hook Protection

Zhi Wang
NC State University
zhi_wang@ncsu.edu

Xuxian Jiang
NC State University
jiang@cs.ncsu.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Peng Ning
NC State University
pning@ncsu.edu

ABSTRACT

Kernel rootkits have posed serious security threats due to their stealthy manner. To hide their presence and activities, many rootkits hijack control flows by modifying control data or hooks in the kernel space. A critical step towards eliminating rootkits is to protect such hooks from being hijacked. However, it remains a challenge because there exist a large number of widely-scattered kernel hooks and many of them could be dynamically allocated from kernel heap and co-located together with other kernel data. In addition, there is a lack of flexible commodity hardware support, leading to the so-called *protection granularity* gap – kernel hook protection requires byte-level granularity but commodity hardware only provides page-level protection.

To address the above challenges, in this paper, we present HookSafe, a hypervisor-based lightweight system that can protect thousands of kernel hooks in a guest OS from being hijacked. One key observation behind our approach is that a kernel hook, once initialized, may be frequently "read"-accessed, but rarely "write"-accessed. As such, we can relocate those kernel hooks to a dedicated page-aligned memory space and then regulate accesses to them with hardware-based page-level protection. We have developed a prototype of HookSafe and used it to protect more than $5,900$ kernel hooks in a Linux guest. Our experiments with nine real-world rootkits show that HookSafe can effectively defeat their attempts to hijack kernel hooks. We also show that HookSafe achieves such a large-scale protection with a small overhead (e.g., around 6% slowdown in performance benchmarks).

Categories and Subject Descriptors D.4.6 [**Operating System**]: Security and protection – Invasive software

General Terms Security

Keywords Malware Protection, Rootkits, Virtual Machines

1. INTRODUCTION

Kernel rootkits are considered one of the most stealthy computer malware and pose significant security threats [28]. By directly subverting operating system (OS) kernels, such rootkits can not only hide their presence but also tamper with OS functionalities to launch various attacks such as opening system backdoors, stealing private information, escalating privileges of malicious processes, and disabling defense mechanisms.

Given the serious security threats, there has been a long line of research on rootkit defense. Specifically, prior research efforts can be roughly classified into three categories. In the first category, systems such as Panorama [33], HookFinder [32], K-Tracer [15], and PoKeR [24] focus on analyzing rootkit behaviors. Systems in the second category are primarily designed for detecting rootkits based on certain symptoms exhibited by rootkit infection. Examples are Copilot [20], SBCFI [21], and VMwatcher [13]. In the third category, systems such as SecVisor [26], Patagonix [16], and NICKLE [23] have been developed to preserve kernel code integrity by preventing malicious rootkit code from executing. Unfortunately, they can be bypassed by *return-oriented* rootkits [11], which will first subvert kernel control flow (i.e., by hijacking function pointers or return addresses on the stack) and then launch the attack by only utilizing legitimate kernel code snippets.

In light of the above threat, it becomes evident that, in addition to the preservation of kernel code integrity, it is also equally important to safeguard relevant kernel control data so that we can preserve the kernel control flow integrity and thus block rootkit infection in the first place. In this paper, we consider kernel data as control data if it is loaded to processor program counter at some point in kernel execution. There are two main types of kernel control data: *return addresses* and *function pointers*. In prior research, there exist extensive studies [1, 2, 9] on how to effectively protect return addresses some of which [1, 9] have been deployed in real-world applications. In this work, our primary focus is to protect those function pointers. Note that function pointers are typically hijacked or "hooked" by rootkits. For ease of presentation, we use the term function pointers and kernel hooks interchangeably.

To safeguard a kernel hook, an intuitive approach [18] is to leverage hardware-based page-level protection so that any write-access to the memory page with the kernel hook can be monitored and verified. This approach will work well if (1) there exist only a very limited number of kernel hooks for protection and (2) these hooks are not co-located together with frequently modified memory data. Unfortunately, in a commodity OS kernel such as Linux and Windows, it is not uncommon that there exist thousands of kernel hooks and these kernel hooks can be widely scattered across the kernel space. Further, many of them might be dynamically allocated from kernel heap and are co-located together with other writable kernel data in the same physical memory frames. If this intuitive approach is deployed, it has to trap all writes to memory pages containing kernel hooks, even those not targeting at kernel hooks. Consequently, it will introduce significant performance overhead, particularly from frequent unnecessary page faults that are caused by write-accesses to irrelevant data. In fact, our investigation with a recent Linux sys-

CCS'09, November 9–13, 2009, Chicago, Illinois, USA.

tem indicates that about 1% of kernel memory writes may cause such unnecessary page faults.

To address the above challenges, in this paper, we present HookSafe, a hypervisor-based lightweight system that is able to efficiently protect thousands of kernel hooks in a guest OS from being hijacked. Our approach recognizes a fundamental challenge, namely the *protection granularity gap*, that hardware provides page-level protection but kernel hook protection requires byte-level granularity. To tackle this challenge, we observe that these kernel hooks, once initialized, rarely change their values. This observation inspires us to relocate kernel hooks to a dedicated page-aligned memory space and then introduce a thin *hook indirection layer* to regulate accesses to them with hardware-based page-level protection. By doing so, we avoid the unnecessary page faults caused by trapping writes to irrelevant data.

We have implemented a prototype of HookSafe based on the latest Xen hypervisor [7] (version 3.3.0) and used it to protect more than 5,900 kernel hooks in a Ubuntu 8.04 Linux system. Our experiments with nine real-world rootkits show that HookSafe can effectively defeat their attempts to hijack kernel hooks that are being protected. We also show that HookSafe achieves such a large-scale protection with only 6% slowdown in performance benchmarks [6, 29]. To the best of our knowledge, HookSafe is the first system that is proposed to enable large-scale hook protection with low performance overhead.

The rest of the paper is structured as follows. We first discuss the problem space HookSafe aims to address in Section 2. Then we present our system design and implementation in Section 3 and Section 4. We show our evaluation results with real-world rootkits and performance benchmarks in Section 5. After discussing limitations of our HookSafe prototype in Section 6, we describe related work in Section 7. Finally, we conclude our paper in Section 8.

2. PROBLEM OVERVIEW

Kernel rootkits can be roughly classified into two categories: Kernel Object Hooking (KOH) and Dynamic Kernel Object Manipulation (DKOM). KOH rootkits hijack kernel control flow while DKOM rootkits do not hijack the control flow but instead subvert the kernel by directly modifying dynamic data objects. In this work, we focus on KOH rootkits since majority of kernel rootkits in the wild are of this type. In fact, a recent thorough analysis [21] on 25 Linux rootkits indicates that 24 (96%) of them make control-flow modifications.

A KOH rootkit can gain the control of kernel execution by hijacking either code hooks or data hooks [31, 32]. Since hijacking a kernel code hook requires modifying the kernel text section which is usually static and can be marked as read-only, it is straightforward to protect them [16, 23, 26]. Kernel data hooks instead are typically function pointers and usually reside in two main kernel memory regions. One is the preallocated memory areas including the data sections and the bss sections in the kernel and loadable kernel modules. The other are the dynamically allocated areas such as the kernel heap. By design, HookSafe aims to prevent kernel rootkits from tampering with kernel hooks in *both* memory regions with low performance overhead.

As mentioned earlier, to efficiently protect kernel hooks, we face a critical challenge of the *protection granularity gap*, where the hardware provides page-level protection but kernel hooks are at the byte-level granularity. Since kernel hooks are scattered across the kernel space and often co-located with other dynamic kernel data, we cannot simply use hardware-based page-level protection.

To better understand it, we have analyzed a typical Ubuntu 8.04 server by using a whole-system emulator called QEMU [22]. Our

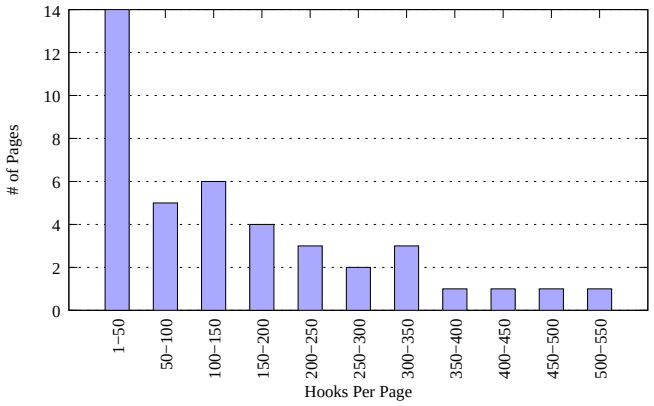


Figure 1: Distribution of 5,881 kernel hooks in a running Ubuntu system

analysis with 5,881 Linux kernel hooks (we describe how we obtain these hooks in Section 5) indicates that they are scattered across 41 physical pages and some of them are located in dynamic kernel heap. In Figure 1 we show the histogram of the number of kernel hooks in a single memory page. We can see that 14 pages contain less than 50 hooks. In the worst case, one hook is allocated in a page (4,096 bytes) along with other 4,092 bytes of dynamic data. As a result, writes to these physical pages would trigger frequent unnecessary page faults if one marked them as write-protected. To quantify these unnecessary page faults, we recorded all kernel memory write operations in the Ubuntu server. Based on the collected log, within a randomly-selected period of 100 seconds, we found that there are in total 700,970,160 kernel memory writes. Among these writes, there is no single write to the set of 5,881 kernel hooks for protection, while the number of writes to the 41 memory pages that contain protected hooks is 6,479,417. In other words, about 1% of kernel memory writes would cause unnecessary page faults and thus introduce expensive switches between a VM and a hypervisor. When designing HookSafe, a key technical task is to avoid these unnecessary page faults while still effectively securing the protected kernel hooks.

In this paper, we assume that a trusted bootstrap mechanism such as tboot [3] is in place to establish the static root of trust of the entire system. With that, a trustworthy hypervisor can be securely loaded which, in turn, can protect the integrity of the guest kernel at boot time. We also assume the runtime integrity of hypervisor is maintained. As such, we consider the attacks to hypervisor including recent SMM ones [14] fall outside the scope of this paper.

3. HOOKSAFE DESIGN

3.1 Overview

HookSafe is a hypervisor-based lightweight system that aims to achieve large-scale protection of kernel hooks in a guest OS so that they will not be tampered with by kernel rootkits. To efficiently resolve the protection granularity gap, in HookSafe we relocate kernel hooks from their original (widely-scattered) locations to a page-aligned centralized location and then use a thin hook indirection layer to regulate accesses to them with hardware-based page-level protection. In other words, we create a shadow copy of the kernel hooks in a centralized location. Any attempt to modify the shadow copy will be trapped and verified by the underlying hypervisor while the regular read access will be simply redirected to

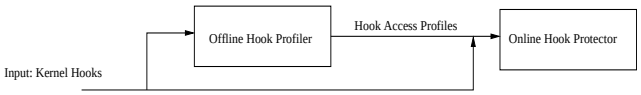


Figure 2: The HookSafe architecture

the shadow copy. By using hook indirection, we avoid the performance overhead caused by trapping legitimate writes to dynamic kernel data around protected hooks.

In HookSafe, all read or write accesses to protected kernel hooks are routed through the hook indirection layer. For performance reasons, we handle read and write accesses differently. Specifically, for a normal write access that updates a kernel hook, because only the hypervisor can write to the memory pages of protected kernel hooks, we will transfer the control from the guest kernel to the hypervisor to commit the update and then switch back to the guest kernel. However, for a read access, we use a piece of indirection code residing in the guest OS kernel memory to read the corresponding shadow hook. By doing so, we avoid the overhead of switching from the guest to the hypervisor and vice versa in read accesses. Note that read accesses to protected hooks could be very frequent, and thus we can benefit significantly by keeping read indirection inside the guest OS kernel.

Figure 2 shows the overall architecture of HookSafe. Given a set of kernel hooks (for protection) as input, HookSafe achieves its functionality in two key steps:

- First, an *offline hook profiler* component profiles the guest kernel execution and outputs a hook access profile for each protected hook. A hook access profile includes those kernel instructions that read from or write to a hook and the set of values assigned to it. In the next step, a hook's access profile will be used to enable transparent hook indirection. For simplicity, we refer to those instructions that access a hook as Hook Access Points (HAPs).

- Second, taking hook access profiles as input, an *online hook protector* creates a shadow copy of all protected hooks and instruments HAP instructions such that their accesses will be transparently redirected to the shadow copy. The shadow hooks are aggregated together in a central location and protected from any unauthorized modifications.

In the rest of this section, we will describe these two steps in detail. We will focus on key ideas in HookSafe's design and defer implementation details to Section 4.

3.2 Offline Hook Profiling

Our first step is to derive, for the given kernel hooks (as input), the corresponding hook access profiles. To do so, there are two main approaches: The first approach is to perform *static analysis* on the OS kernel source code and utilize known program analysis techniques such as points-to analysis [5] to automatically collect hook access profiles. The second approach is to leverage *dynamic analysis* without the need of requiring the OS kernel source code. In particular, dynamic analysis runs the target system on top of an emulator (e.g., QEMU [22]) and monitors every memory access to derive the hook access instructions. In comparison, dynamic analysis allows for recording precise runtime information such as the values a hook has taken, but potentially has less coverage while static analysis is more complete but less precise.

We note that both approaches have been widely explored before and the results can be directly applicable in HookSafe. In our current prototype, we have chosen to implement the offline hook profiler based on dynamic analysis.

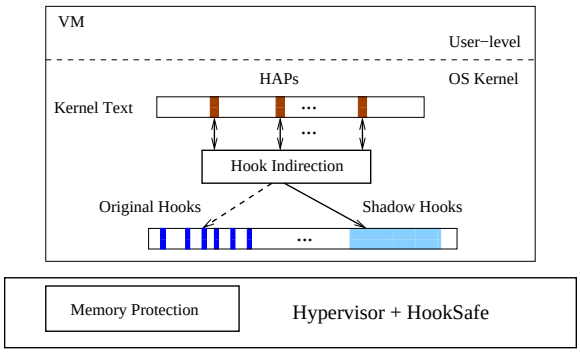


Figure 3: The architecture of online hook protection.

3.3 Online Hook Protection

After collecting hook access profiles, our next step is to efficiently protect kernel hooks in a guest OS from being manipulated by kernel rootkits. Figure 3 shows the architecture of our online hook protection. The essential idea here is to leverage a thin hook indirection layer to regulate accesses to kernel hooks. Specifically, after a guest OS boots up, we first create a shadow copy of identified hooks, and then instrument all HAPs in kernel code so that read or write accesses will be redirected to the hook indirection layer. In addition, there exists a memory protection component in the hypervisor that protects the indirection code, the shadow hooks, and the kernel code in the guest OS from being tampered with. Next, we will describe how HookSafe initializes online hook protection and how it handles read/write accesses.

3.3.1 Initialization

HookSafe initializes the online hook protection in two steps. It first uses an in-guest short-lived kernel module to create the shadow copy of kernel hooks and load the code for the indirection layer. Then it leverages the online patching provided by the hypervisor to instrument the HAPs in the guest kernel. As mentioned earlier, we assume the system bootstrap process and the in-guest kernel module loading are protected by trusted booting [3]. Next we describe these two steps in detail.

As a part of system bootstrap process, the in-guest kernel module will be loaded to allocate memory pages from the non-paged pool. The non-paged memory allocation is needed to prevent them from being swapped out. After that, the kernel module will copy protected kernel hooks at their original locations to the newly allocated memory pages. Then it will load the code of the indirection layer to these memory pages. Before the guest kernel module unloads itself, it will make a hypercall to notify the hypervisor the starting address and size of the memory pages so that they can be protected.

When the hypervisor receives the hypercall regarding the guest memory pages for shadow hooks and indirection code, it conducts the second step to patch HAPs in the kernel code. The basic idea is to detour the execution of an HAP instruction to the hook indirection layer so that the access to the original kernel hook will be redirected to the corresponding shadow copy.

3.3.2 Run-Time Read/Write Indirection

After the initialization, all accesses to protected hooks at the HAPs will be redirected to the hook indirection layer. It then han-

dles hook accesses differently depending on whether it is a read or write access. For read accesses, the indirection layer simply reads from the shadow hooks then returns to the HAP site. For write accesses, the indirection layer will issue a hypercall and transfer the control to the hypervisor. Then the memory protection component in the hypervisor will validate the write request and update the shadow hook if the request is valid. To validate a write request, HookSafe requires the new hook value to be seen in the offline profiling phase. We note that other policies can also be naturally supported in HookSafe, including those proposed in SBCFI [21] to check if the new kernel hook points to a valid code region, a valid function, a valid function with correct type, or the related points-to set calculated from static analysis.

We point out that the hypervisor-based memory protection component protects not only the centralized shadow hooks but also the code for hook indirection and other legitimate kernel code. To do so, HookSafe utilizes the shadow page table realized in the hypervisor for a running guest VM and sets proper protections for memory pages of the protected contents (more in Section 4.3).

3.3.3 Run-Time Tracking of Dynamically Allocated Hooks

The design of HookSafe is complicated by the support of dynamically allocated hooks. In particular, since those hooks are allocated and deallocated at runtime, we need to effectively keep track of these events. To this end, we notice that a dynamically allocated hook is typically embedded in a dynamic kernel object. In other words, a dynamic hook will be created when the hosting kernel object is being allocated and instantiated from the kernel heap. Similarly, a dynamic hook will be removed when the hosting kernel object is being de-allocated and the related memory space is being returned back to kernel heap. Accordingly, we can instrument the memory allocation/deallocation functions and utilize the run-time context information to infer whether a particular kernel object of interest is being allocated or de-allocated. If one such kernel object that contains a kernel hook is being allocated, a hypercall will be issued to HookSafe to create a shadow copy of the hook (not the entire hosting kernel object!). Similarly, another hypercall is triggered to remove the shadow copy when the hosting kernel object is released. By introducing two extra hypercalls, HookSafe is able to track the creation and removal of dynamically allocated kernel hooks. After that, HookSafe's hook indirection layer works the same regardless of the nature of kernel hooks.

Another related challenge is the recognition of those dynamically allocated kernel hooks that may already exist before loading HookSafe's in-guest module for hook protection. To address that, a natural approach is to initiate the run-time tracking immediately after the guest OS begins execution. However, it implies HookSafe needs to instrument the memory management functions at the very first moment when the guest OS executes. Also without the help from an in-guest module, HookSafe needs to maintain its own buffer to record those dynamically-allocated kernel hooks. To avoid that, in our prototype, we instead take another approach. Specifically, our approach exploits the fact that any dynamically allocated kernel object must be accessible in some way from certain global variable(s) or CPU register(s). If one imagines kernel objects as a graph where the edges are pointers, then all objects will be transitively reachable from at least one global variable. If an object is not reachable in this way, then the kernel itself will not be able to access it and the object cannot be used. A similar observation has also been made in previous work on both garbage collection and state-based control-flow integrity [21]. As a result, with the knowledge of these global variables, it is a straightforward process to identify the current run-time addresses of target kernel objects, which contain kernel hooks. After that, we can apply the normal process of creating shadow copies of these pre-existing kernel hooks and patching the guest kernel to redirect access to their shadow copies.

3.4 Hardware Register Protection

In addition to regular memory-based kernel hooks, hardware registers such as *GDTR, IDTR, DR0-DR7* debug registers, and *SYSENTER MSR* registers can also be potentially exploited by rootkits to hijack kernel control flow. These hooks are special in that they are invoked directly by the hardware. Therefore it is vital for HookSafe to regulate accesses to these registers as well. To do that, we use hardware-based virtualization support to intercept and validate any write attempts to these registers. Related to the hardware register protection is how we secure the *GDT* and *IDT* descriptor tables. These two tables contain critical system data structures and their contents must be protected. We protect these two tables using the hardware-based page-level protection.

Another related issue is how we prevent DMA from being abused to subvert the HookSafe's memory protection. In particular, to limit the physical memory accessible to a DMA engine of an I/O device, we utilize the hardware-based IOMMU support (already implemented in Xen) in the recent CPU/chipsets to map the address space of DMA engine to a safe place. In other words, the way the IOMMU is set up preludes the possibility of overwriting HookSafe as well as HookSafe-protected memory regions in a guest.

4. IMPLEMENTATION

We have implemented a prototype of HookSafe. The online hook protection component was developed based on the Xen hypervisor [7] (version 3.3.0), and the offline hook profiling component is based on QEMU [22], an open source whole-system emulator. The current prototype is mainly implemented and evaluated in a system running Ubuntu Linux 8.04. Because of that, for most of this section, we choose it as the default guest OS protected by HookSafe. In the following, we will first present the implementation of the offline hook profiler. Then we will describe how we implement the hook indirection and the memory protection.

4.1 Offline Hook Profiler

Our offline hook profiler is essentially a whole-system emulator with additional functionality in instrumenting and monitoring the execution of every memory access instruction. In particular, given a list of kernel hook locations for protection, we run the target system on top of the profiler. At the end of a run, the profiler records, for each kernel hook, a list of HAP instructions that read from or write to these kernel hooks and the set of values the hook may take at runtime.

QEMU implements a key virtualization technique called binary translation that rewrites the guest's binary instructions. Our prototype extends the binary translator in QEMU with additional instrumentation code to record executions of instructions that read or write memories. If an instruction accesses any kernel hook in the given list, we mark it as an HAP instruction and log the value that is written to or read from the hook. For a dynamically allocated kernel hook, the profiler also tracks the creation of the hosting kernel object and locates the runtime hook location. At the end of profiling, the collected HAP instructions and recorded hook values will be compiled as the corresponding hook access profile.

Figure 4 shows an example profile for the kernel hook *ext3_dir_operations ->readdir*, which is located in the *ext3.ko* loadable kernel module (LKM) and has been hijacked by existing kernel rootk-

```
Hook:  ext3_dir_operations->readdir (0xc025c924)
================================================

HAP1 (Access Type: READ):
        address:  0xc015069a (vfs_readdir+0x1D)
    instruction:  83 78 18 00 cmpl $0x0,0x18(%eax)
        content:  0xc016f595 (ext3_readdir)

HAP2 (Access Type: READ):
        address:  0xc01506dd (vfs_readdir+0x60)
    instruction:  ff 53 18    call *0x18(%ebx)
        content:  0xc016f595 (ext3_readdir)
```

Figure 4: An example access profile related to *ext3_dir_operations ->readdir* kernel hook

its (Section 5) for hiding purposes. The profiled results indicate that this specific hook is accessed by two instructions located at $0xc015069a$ and $0xc01506dd$ (both in the *vfs_readdir* function). During the entire profiling, this particular hook always points to the *ext3_readdir* function and there is no instruction observed that will update the hook value.

4.2 Hook Indirection

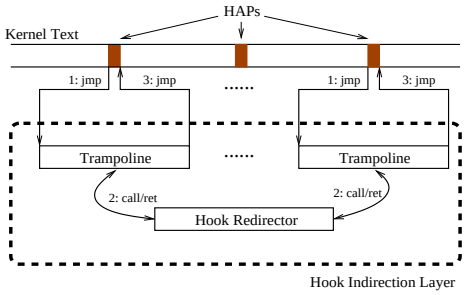


Figure 5: The implementation of hook indirection

The key novelty in HookSafe is to instrument every HAP instruction such that the access to the (original) kernel hook is transparently redirected to the corresponding shadow hook. In Figure 5, we show how the hook indirection is realized in our prototype. In essence, the hypervisor replaces the HAP instruction at runtime with a *jmp* instruction to detour the original execution flow to specially-crafted trampoline code. The trampoline code collects runtime context information that will be used by the hook redirector to determine the exact kernel hook being accessed. After the hook redirector processes the actual read or write to a shadow hook, the trampoline will execute the HAP-specific overwritten instructions, if any, before returning back to the original program. To make our prototype memory-efficient, each HAP instruction has its own copy of trampoline code but the hook redirector code is shared by all the HAPs. Note that both trampoline and hook redirector code reside in the guest kernel memory and are protected by the hypervisor's memory protection component. In Figure 5, the arrowed lines (with instructions and numbers on them) show the control flow transfers among HAPs, trampolines and the hook redirector.

4.2.1 HAP Patching

Our implementation uses a five-byte *jmp* instruction (one byte opcode $0xe9$ plus the 32-bit offset as operand) to detour the control flow from an HAP instructions to its trampoline code in the hook indirection layer. Since x86 architecture uses instructions with variable lengths, we must align the overwritten bytes to the instruction boundary. When an HAP instruction occupies more than five bytes, we will fill the rest space with *NOP* instructions. When an HAP

```
vfs_readdir in linux/fs/readdir.c:
  c0150693:   8b 46 10      mov    0x10(%esi),%eax
  c0150696:   85 c0         test   %eax,%eax
  c0150698:   74 62         je     c01506fc <vfs_readdir+0x7f>
  c015069a:   83 78 18 00   cmpl   $0x0,0x18(%eax)
  c015069e:   74 5c         je     c01506fc <vfs_readdir+0x7f>
```

(a) An example HAP instruction (highlighted in bold font)

```
vfs_readdir in linux/fs/readdir.c
  if(!file->f_op ||!file->f_op->readdir)
        goto out;
```

(b) The C source code related to the HAP (highlighted in bold font)

Figure 6: An example HAP instruction and the related C code

instruction has less than five bytes, we overwrite the subsequent instructions to make the space for the five-byte *jmp* instruction, and execute these overwritten instructions at the end of hook indirection. By doing so, our detouring code preserves the boundary of subsequent instructions.

Using the same example in Figure 4, we show the first HAP instruction (located in $0xc015069a$) and its surrounding instructions in Figure 6(a). The corresponding C source code is shown in Figure 6(b), which belongs to the *vfs_readdir* function defined in *linux/fs/readdir.c* of the linux kernel. In the disassembled code, the first three instructions test whether *file->f_op* is NULL. The fourth instruction is the HAP instruction that checks whether the hook *file->f_op->readdir* is NULL. As shown in Figure 6(a), this particular HAP instruction only occupies four bytes. To successfully detour its execution with the five-byte *jmp* instruction, we need to overwrite the subsequent instruction located at $0xc015069e$ with two bytes as well.

In the patching process, there are two caveats that we have experienced in our prototype, particularly when overwriting additional instructions that follow the patched HAP instruction. First, if two HAPs are close to each other, the first detouring *jmp* instruction may overwrite the second HAP instruction. Second, even worse, some instruction overwritten by the *jmp* instruction may be a jump target. In the example shown in figure 6(a), there may exist an instruction that directly jumps to the second je instruction, which unfortunately has been overwritten by the detouring *jmp* instruction. In other words, that instruction would essentially jump to the middle of the detouring *jmp*, which typically causes an invalid opcode exception.

These two scenarios occur because other instructions than the original HAP instruction are overwritten by the detouring *jmp* instruction. We solve this problem by conducting function-level binary rewriting similar to Detours [12]. Instead of locally modifying the original function, we create a new copy of it in which we replace HAP instructions with *jmp* instructions and shift the subsequent instructions accordingly. In addition, we replace the first instruction in the old function with a *jmp* instruction so that any function call to it will be redirected to the new function. In this way, we avoid rewriting the entire kernel image. Note that we assume there is no control transfer from one function to the middle of another function, which was empirically confirmed in our evaluation.

4.2.2 Read/Write Indirection

The hook indirection layer in HookSafe has two components: trampoline and redirector. The trampoline code prepares the hook-related context information (e.g., the HAP address and machine registers' contents). The redirector uses the context information to find which hook is being read or written and then identify the corresponding shadow hook.

For the support of variable-length instructions, the HAP patching may overwrite additional instructions that follow an HAP instruction. To reclaim them, our prototype customizes the trampoline code for each detoured HAP instruction as follows. First, the additional overwritten instruction(s) are appended to the end of the trampoline code. By doing so, they will be executed when the control returns back from the hook redirector. Second, at the end of the trampoline code, we further append an additional *jmp* instruction so that the control is transferred back to the original program.

Upon the call from the trampoline code, the redirector first determines which hook is being accessed based on the current CPU context and the semantics of the detoured HAP instruction. Using the first HAP instruction in Figure 6(a) as an example the hook's address is the register eax plus $0x18$. The redirector retrieves the content of register eax from the CPU context saved by our trampoline code and determines the original kernel hook that is being accessed. After that, the redirector identifies the corresponding shadow hook and performs the desired access indirection. If it is a read access, the redirector will read the shadow hook and update the saved CPU states to reflect the effect of HAP instruction. Continuing the previous HAP instruction example (Figure 6(a)), the redirector will update the $eflags$ register in the saved CPU states to indicate whether *file->f_op->readdir* is $NULL$. After returning from the redirector, the trampoline will restore saved CPU states. By doing so, the read access of the original kernel hook is effectively redirected to its shadow copy. If it is a write access, the redirector will make a hypercall to the hypervisor so that the memory protection component can verify the new hook value and update the shadow hook if it is legitimate.

In our prototype, instead of completely ignoring original kernel hooks, we also utilize them to detect rootkits' hooking behavior. More specifically, for each redirected hook read access, the hook indirection layer in addition performs a consistency check between the original kernel hook and its shadow copy. Any difference would indicate that the original hook has been compromised. Similarly, for each redirected hook write access, if the write operation is legitimate, we need to update both the shadow hook and the original hook to keep them synchronized.

4.2.3 Run-Time LKM and Hook Tracking

To support runtime tracking of dynamically allocated kernel hooks, we first observe that the lifetimes of these hooks are consistent with their hosting kernel objects. In addition, in Linux, these hosting kernel objects are typically allocated/deallocated through the SLAB interface. More specifically, the SLAB allocator manages caches of objects (that allow for fast and efficient allocations) and each different type of object has its own SLAB cache identified by a human-readable name. Since there are two main functions to allocate and deallocate a kernel objects, i.e., $kmem\_cache\_alloc$ and $kmem\_cache\_free$, our current prototype instruments these two functions using a very similar technique with hook indirection. Specifically, before these two functions return, the instrumented code checks whether the SLAB manages a particular kernel object of interest (i.e., whether it contains a kernel hook that needs to be shadowed). If so a hypercall will be issued to notify HookSafe so that it can track the hook creation and termination. Note that the instrumented code runs in the guest kernel space and the world switch only occurs when the kernel object being allocated/deallocated through the SLAB interface contains a kernel hook of interest.

Related to dynamic kernel hook tracking is how we support the kernel hooks inside the Loadable Kernel Modules (LKMs). For a given LKM, because its runtime memory will not be determined until at runtime, it poses additional challenges for HookSafe to precisely determine the locations of those hooks contained in the LKM. Fortunately, for those kernel hooks (and HAP instructions) inside a LKM, the relative offsets to the LKM base address where the module is loaded are fixed. Therefore a LKM hook's runtime location can be simply calculated as the addition of the LKM's current base location and the relative offset. Based on this observation, a hook address is extended to a tuple of (*module_hash*, *hook_offset*), where the hash is used to uniquely identify a module.

To precisely locate the LKM base address at runtime, we utilize the well-known technique called virtual machine introspection [10, 23, 26]. Specifically, during our profiling and online HookSafe protection, we leverage the virtual machine introspection to intercept module loading/unloading events in the guest kernel. After a module is loaded and fixed up by the guest kernel, we then derive the runtime location of a LKM hook by simply adding the base address of the module and the relative offset of the hook. An example of such LKM hook is shown in Figure 4 where the *ext3_dir_operations ->readdir* is a kernel hook inside the *ext3.ko* module.

4.3 Memory Protection

To protect the guest kernel code and the in-guest memory used by HookSafe, we leverage the shadow page table (SPT) management subsystem in the Xen hypervisor. In the SPT memory management mode, the hypervisor maintains an SPT for each guest, which regulates the translation directly from a guest virtual address to the host physical address. Any update to the guest page table (GPT) in the guest kernel is trapped and propagated to the SPT by the hypervisor. In other words, the hardware performs the address translation solely with the shadow page table.

To protect guest memories, we check if the guest virtual address is in the range of our protected memories before propagating the changes in GPT to SPT. If so, we make sure the physical pages are marked read-only.

4.4 System Call Indirection Optimization

When building our prototype, we realized that the system call table provides a unique opportunity for optimization. Note that the Linux's system call table shares a memory page with other dynamic data [23], which means that we cannot simply mark the memory page read-only but rely on HookSafe's protection. In the meantime, a system call table usually contains a large number of hooks. For instance, in the kernel we tested, the system call table has 330 function pointers and each of them may be hooked by a rootkit. Due to the fact that accesses to the system call table is fairly frequent, it is also critical to optimize its indirection to reduce the overhead.

We observe that the system call table in the linux kernel is accessed by only two *read* HAP instructions, and both of them has the base address of the system call table hardcoded in the instructions. This provides a unique opportunity for optimization. In our prototype, instead of creating a shadow for each individual system call hook, we simply create a shadow system call table and replace the base addresses in the two HAP instructions with the new shadow table address. By doing so, we essentially eliminate the system call redirection overhead caused by hook indirection. Our experience indicates that this is a special case and we did not find any other hooks that can be optimized this way.

5. EVALUATION

In this section, we present our evaluation results. In particular, we have conducted two sets of experiments. The first set of experiments (Section 5.1) is to evaluate HookSafe's effectiveness in

Rootkit	Attack Vector	Hooking Behavior	HookSafe Results	
			Outcome	Reason
adore-ng 0.56	LKM	proc_root_inode_operations ->lookup	Hiding fails	Hook indirection
		proc_root_operations ->readdir	Hiding fails	Hook indirection
		ext3_dir_operations ->readdir	Hiding fails	Hook indirection
		ext3_file_operations ->write	Hiding fails	Hook indirection
		unix_dgram_ops ->recvmsg	Hiding fails	Hook indirection
eNYeLKM 1.2	LKM	kernel code modification	Installation fails	Memory Protection
sk2rc2	/dev/kmem	sys_call_table[__NR_oldolduname]‡	Installation fails	Memory Protection
superkit	/dev/kmem	sys_call_table[__NR_oldolduname]‡	Installation fails	Memory Protection
Phalanx b6	/dev/mem	sys_call_table[__NR_setdomainname]‡	Installation fails	Memory Protection
mood-nt 2.3	/dev/kmem	sys_call_table[__NR_olduname] ‡	Installation fails	Memory Protection
override	LKM	sys_call_table[__NR_getuid]†	Hiding fails	Hook indirection
		sys_call_table[__NR_geteuid]†	Hiding fails	Hook indirection
		sys_call_table[__NR_getdents64]†	Hiding fails	Hook Indirection
		sys_call_table[__NR_chdir]†	Hiding fails	Hook indirection
		sys_call_table[__NR_read]†	Hiding fails	Hook indirection
Sebek 3.2.0b	LKM	sys_call_table[__NR_read]‡	Installation fails	Memory Protection
hideme.vfs	LKM	kernel code modification	Installation fails	Memory Protection

Table 1: Effectiveness of HookSafe in preventing 9 real world kernel rootkits: *Hiding fails* indicates that although the rootkit modified the original kernel hooks, it failed to hijack the control flow because HookSafe has redirected the hook to its shadow; *Installation fails* indicates that the rootkit hooking behavior causes the memory protection violation, hence failing the installation. Additionally, depending on how the rootkit searches for the system call table, it may locate either the original system call table (marked with †) or the shadow system call table (marked with ‡).

preventing real-world rootkits from tampering with kernel hooks. We tested HookSafe with nine real-world rootkits. It successfully prevented all of them from modifying protected hooks and hiding themselves. The second set of experiments (Section 5.2) is to measure performance overhead introduced by HookSafe. We evaluated HookSafe on benchmark programs (e.g., UnixBench [29] and ApacheBench[6]) and real-world applications. Our experimental results show that the performance overhead introduced by HookSafe is around 6%.

In our experiments, HookSafe takes as input two sets of kernel hooks. The first set includes $5,881$ kernel hooks in preallocated memory areas of main Linux kernel and dynamically loaded kernel modules. Specifically, we derive this set by scanning the data/bss sections of the kernel and LKMs in a guest VM running Ubuntu Server 8.04 (with a default installation). In our experiment, we examine those sections every four bytes (32-bit aligned) and consider it as a kernel hook if it points to the starting address of a function in the kernel or kernel modules. At the end, we found $5,881$ kernel hooks in the guest VM. The second set is from 39 kernel objects (with function pointers) that will be dynamically allocated from kernel heap. We obtain this set by manually going through a subset of the entire Linux source code and locate those kernel objects of interest. Note that during runtime, it is not uncommon that tens or hundreds of copies of the same type of kernel objects will be allocated. Not surprisingly, a large part of them are related to timer, callback/notifier functions, and device drivers. We point out that this set can be further improved from existing systems such as [15, 24, 32, 33] as they are capable of profiling rootkit execution and reporting those compromised kernel objects with hooks. Given the above two sets of kernel hooks, with our offline profiler, we identified 968 HAP instructions for these hooks: 785 of them are read HAPs while the remaining 183 are write HAPs. Next we describe our experiments in detail.

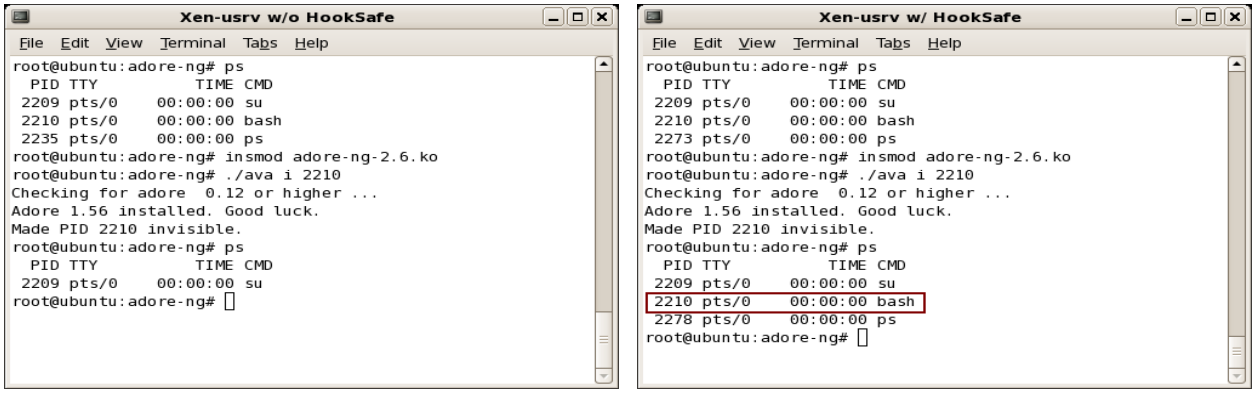
5.1 Effectiveness Against Kernel Rootkits

We have evaluated HookSafe with nine real-world Linux 2.6 rootkits shown in Table 1. These rootkits cover main attack vectors and hooking behaviors of existing kernel rootkits, therefore providing a good representation of the state-of-the-art kernel rootkit technology. HookSafe successfully prevented these rootkits from modifying the protected kernel hooks: these rootkits either failed to hide their presences or failed to inject code into the kernel. In the following, we describe in detail our experiments with two representative rootkits.

***Adore-ng* Rootkit Experiment** The *adore-ng* rootkit infects the kernel as a loadable kernel module. If successfully installed, it will hijack a number of kernel hooks and gain necessary control over kernel execution so that it can hide certain rootkit-related files, processes, and network connections. Meanwhile, it also has a user-level control program named *ava* that can send detailed instructions (e.g., hiding a particular file or process) to the rootkit module.

For comparison, we performed our experiments in two scenarios: First, we loaded *adore-ng* in a guest OS that is *not protected* by HookSafe and showed that it can successfully hide a running process as instructed by the *ava* program (see Figure 7(a)). Second, we repeated the experiment in the same guest OS that is now *protected* by HookSafe. This time the rootkit failed to hide the process (see Figure 7(b)). By analyzing the experiments, we found that the rootkit was able to locate and modify certain kernel hooks at their original locations. However, since now control flows related to these hooks are determined by the shadows hooks, this rootkit failed to hijack the control flow and thus was unable to hide the running process.

As mentioned earlier, the hook indirection layer performs an additional check by comparing the original kernel hook with its shadow copy whenever the hook is accessed. As a result, we are able to successfully identify these kernel hooks that are being manipulated by *adore-ng*. Our experiments show that the *adore-ng* rootkit hijacks a number of kernel hooks, including *proc_root_inode_operations ->lookup*, *proc_root_operations ->readdir*, *ext3_dir_operations ->readdir*, *ext3_file_operations ->write*, and *unix_dgram_ops ->recvmsg*. As pointed out earlier, the *ext3_file_operations* kernel object is a part of the *ext3.ko* and this module will be loaded somewhere in the kernel heap at run time. A detailed analysis with the rootkit source code reveals that the first two are hijacked for hiding processes, the next two are for hiding files and directories, and



(a) Adore-ng hides the *bash* process (b) HookSafe prevents adore-ng from hiding processes

Figure 7: HookSafe foils the adore-ng rootkit.

the last one is for filtering messages to the *syslogd* daemon. This demonstrates that, with HookSafe's protection, this rootkit and others of its kind will not be able to hijack these hooks to hide their malicious activities.

***Mood-nt* Rootkit Experiment** The *mood-nt* rootkit attacks the Linux kernel by directly modifying the kernel memory through the */dev/kmem* interface. This is a rather advanced kernel-level attack that leverages an attack strategy reflecting the so-called *return-to-libc* attacks [8, 11, 27]. Specifically, it first overwrites a function pointer, i.e., a system call table entry called *__NR_olduname*, via the */dev/kmem* interface and makes it pointing to a kernel function called *vmalloc*. Then it "executes" this function by invoking a syscall (with syscall number *__NR_olduname*) and this legitimate kernel function will allocate a chunk of contiguous kernel memory for the rootkit. After that, it populates the memory with its own code (via the same */dev/kmem* interface), which essentially installs itself in the kernel. Next it overwrites the same function pointer again but this time it will point to its own code, and invokes it by making another syscall (with the same syscall number). Finally, it launches its payload and starts manipulating the OS kernel.

The *mood-nt* rootkit demonstrates the trend of leveraging the *return-to-libc* attack strategy to evade systems [16, 23, 26] that aim to preserve kernel code integrity. Instead of hijacking a syscall table entry, it can also choose to control any other kernel hook and potentially make use of it multiple times and each time it can point to an arbitrary kernel function routine or code snippet. By doing so, the rootkit can manage to sequentially execute these legitimate kernel functions in the order chosen by the rootkit *without* injecting its own code.

With HookSafe's protection, this rootkit was stopped at the first step when it tried to overwrite the function pointer *__NR_olduname* in the shadow syscall table. Interestingly, the rootkit somehow is able to locate and attempt to modify the corresponding shadow copy of the function pointer. It turns out it identifies the shadow copy by following the new base address of the syscall table in the syscall handler, which is provided in our optimization (Section 4.4). As a result, by stopping *mood-nt* at the first place, HookSafe essentially prevents function pointers from being subverted to manipulate kernel control flow.

5.2 Performance

To evaluate the performance overhead introduced by HookSafe, we measured its runtime overhead on 10 tasks including those in the UnixBench [29] and Linux kernel decompression and compilation. We also measured its throughput degradation on a web server using the ApacheBench [6].

Item	Version	Configuration
Ubuntu Server	8.04	standard installation (kernel 2.6.18-8)
Kernel Build	2.6.18-8	make defconfig & make
Gunzip	1.3.12	tar -zxf <file>
Apache	2.2.8	default configuration
ApacheBench	2.0.40-dev	-c3 -t 60 <url/file>
UnixBench	4.1.0	default configuration

Table 2: Software package configurations for evaluation

Our testing platform is a Dell Optiplex 740 with an AMD64 X2 5200+ CPU and 2GB memory. HookSafe ran with the Xen hypervisor of version 3.3.0. The guest OS is a default installation of Ubuntu server 8.04 with a custom compile of the standard 2.6.18-8 kernel. Table 2 lists the configurations of the software packages used in our evaluations. In the Apache test, we ran an Apache web server to serve a web page of 8K bytes. We ran the ApacheBench program on another machine in the same gigabit Ethernet to measure the web server throughput. For each benchmark, we ran 10 experiments with and without HookSafe and calculated the average.

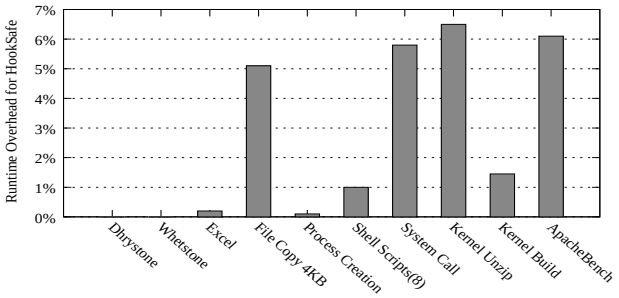


Figure 8: Runtime overhead on UnixBench and three application benchmarks for HookSafe.

In Figure 8 we show the runtime overhead on 7 tasks in the UnixBench, kernel decompression and kernel build, as well as the throughput drop in ApacheBench. We can see that the maximum overhead added by HookSafe is 6.5% when the standard Linux kernel source package *linux-2.6.18-8.tar.gz* (51 megabytes) is decompressed with the *gunzip* program. In the ApacheBench test, the throughput degradation caused by HookSafe is 6.1%.

The ApacheBench experiment is particularly interesting. In a short time period of 1 minute, the Apache server accepted more than 10,000 TCP connections from the ApacheBench. For each TCP connection, two dynamic kernel objects containing function pointers were created: *struct sock* (a kernel object for an active net-

work socket) and *struct ip_conntrack* (a kernel object used in the Linux packet filtering framework for IP connection tracking purposes). Note that although the *sock* object will be destroyed when the connection is closed, the *ip_conntrack* object lives longer until the connection completely expires. During the ApacheBench test, at its peak, there were 24 active sockets and $10,096$ ip_conntrack objects alive, which means during this particular test, there are around $16,000$ kernel hooks (including more than $10,120$ dynamically allocated hooks) being protected by HookSafe. To quantify the performance overhead, we measured the ApacheBench's throughput by only enabling run-time hook tracking. The results show that the Apache server is able to transmit $491.81MB$ in one minute period, an overhead of 0.7% when compared to the throughput of $495.39MB$ without HookSafe's protection.

In addition, we performed a micro-measurement on the overhead due to the hypercall made when creating a socket object. In this measurement, we recorded the hardware timestamp counter register (with the instruction *rdtsc*) [4] right before and after the socket creation call, i.e., the *sk_alloc* function. Originally, it took 2266 CPU ticks for sk_alloc to complete while, with the overhead of an additional hypercall, it required 5839 CPU ticks to complete. As a result, the hypercall incurs 157% overhead for a socket creation. Note that this overhead occurs only when the socket is being created, which counts for a small part of the computation for the entire lifetime of a socket connection. In other words, the performance degradation due to these hypercalls is amortized over the lifetime of kernel objects. The longer the lifetime, the smaller the performance overhead.

In conclusion, HookSafe is lightweight and able to achieve large-scale hook protection with around 6% performance slowdown.

6. DISCUSSION

A fundamental limitation faced by our current HookSafe prototype for Linux guest OS is that hook access profiles are constructed based on dynamic analysis and thus may be incomplete. The lack of completeness is an inherent issue of dynamic analysis shared by other approaches [15, 24, 31, 32]. In our case, it could result in missing HAPs and legitimate hook values and has the following impact on the protected system. First, since HookSafe maintains the consistency between original hooks and shadow hooks, unpatched read HAP instructions would work normally in a clean system by reading from original hook locations. However, if a guest OS is compromised, there may exist a (small) time window in which a rootkit may hijack the control flow. Specifically, after a rootkit modifies original hooks and before HookSafe detects the inconsistency, the control flow will be hijacked if an unpatched HAP instruction reads from the original hook location to decide a control transfer. Second, for unpatched write HAP instructions, their writes to original hook locations will not affect patched read HAPs. Before HookSafe detects the inconsistency, the protected system will be in an unstable state. Third, for legitimate hook values missing in hook access profiles, HookSafe will raise a false alarm when an HAP instruction attempts to update a hook with such a value. However this is not a serious problem during our experiments. We detected only 5 (or 0.085%) kernel hooks with an incomplete hook value set.

To mitigate the incompleteness problem, there are two possible approaches. The first one is to improve the coverage of dynamic analysis. Recent efforts such as multiple path exploration [17] have shown promise that one can leverage runtime information to systematically guide the exploration process to achieve better coverage. The second one is to combine a complementary approach – static analysis (Section 3.2), which is more complete but less precise. Note that an imprecise hook access profile has the implication of causing false alarms. However, the integration of dynamic analysis and static analysis remains an interesting research problem.

Another limitation is that HookSafe assumes the prior knowledge of the set of kernel hooks that should be protected from rootkit attacks. In other words, HookSafe itself is not designed to automatically discover those kernel hooks. From another perspective, this problem is being addressed to some extent by a number of existing systems, such as HookFinder[32] and HookMap [31], to systematically derive the set of kernel hooks that are or will be of interest to rootkits. We expect that our work will be combined with these efforts in the future to enable integrated hook discovery and rootkit defense.

7. RELATED WORK

Kernel Rootkit Prevention The first area of related work includes recent systems that aim at preventing kernel rootkit infection. For example, SecVisor [26] is a tiny hypervisor that uses hardware support to enforce kernel code integrity. Patagonix [16] provides hypervisor support to detect covertly executing binaries. NICKLE [23] mandates that only verified kernel code will be fetched for execution in the kernel space. However, these systems do not protect kernel hooks from being subverted to compromise kernel control flow, which is the main goal of HookSafe.

Lares [18] is a closely related work and our work mainly differs from it in two ways. First, our goal is to enable efficient, large-scale hook protection while Lares is mainly intended to enable reliable active monitoring of a VM by securing the execution path in which a monitoring point has been planted. Second, Lares directly uses hardware-based page-level protection to trap all writes to those memory pages containing kernel hooks. As a result, any write to irrelevant dynamic data but in the same physical page will cause a page fault (Section 2). When it is applied to protect thousands of hooks that are often co-located with dynamic kernel data, it will lead to significant performance overhead. In comparison, HookSafe recognizes the protection granularity gap and solves it by introducing a hook indirection layer that relocates protected kernel hooks to a page-aligned centralized memory space. By doing so, our approach only incurs small (6%) performance overhead.

Kernel Rootkit Detection The second category of related work aim at detecting rootkit presence. For example, a number of rootkit detection tools such as System Virginity Verifier [25] validate the kernel code and examine the kernel data (including hooks) known to be the targets of current rootkits. Copilot[20] uses a trusted add-in PCI card to grab a runtime OS memory image and infers possible rootkit presence by detecting any kernel code integrity violations. The approach is extended by follow-up research to examine other types of violations such as kernel data semantic integrity [19] and state-based control flow integrity [21]. Livewire [10] pioneered the virtual machine introspection methodology to inspect the inner states of guest VM to detect malware infections. Other systems such as Strider GhostBuster [30] and VMwatcher [13] leverage the self-hiding nature of rootkits to infer rootkit presence by detecting discrepancies between the views of a system from different perspectives. Note that all these approaches were proposed to detect kernel rootkits after the system is infected. In comparison, HookSafe targets at preventing rootkits by protecting kernel hooks and kernel text from being manipulated by them.

Kernel Rootkit Analysis There also exist a number of recent efforts [15, 24, 31, 32, 33] on analyzing and profiling kernel-mode malware, especially kernel rootkits. The goal of these efforts is to enrich our understanding on stealthy malware, including their hooking behavior and targeted kernel objects. Specifically,

Panorama [33] performs system-wide information flow tracking to understand how sensitive data (e.g., user keystrokes) are stolen or manipulated by malware. HookFinder [32] applies dynamic tainting techniques to identify and analyze malware's hooking behavior. HookMap [31] monitors normal kernel execution to identify potential kernel hooks that rootkits may hijack for hiding purposes. K-Tracer [15] makes a step further to systematically discover system data manipulation behavior by rootkits. PoKeR [24] defines four key aspects of kernel rootkit behaviors and accordingly proposes a combat tracking approach to efficiently characterize and profile them. HookSafe has a different goal: instead of locating the kernel hooks hijacked by rootkits, it is designed to protect them from being hijacked. Therefore, HookSafe is complementary to the above systems and thus can be naturally integrated together with them.

8. CONCLUSION

We have presented the design, implementation, and evaluation of HookSafe, a hypervisor-based lightweight system that can protect thousands of kernel hooks from being hijacked by kernel rootkits. To enable large-scale hook protection with low overhead, HookSafe overcomes a critical challenge of the *protection granularity gap* by introducing a thin hook indirection layer. With hook indirection, HookSafe relocates protected hooks to a continuous memory space and regulates accesses to them by leveraging hardware-based page-level protection. Our experimental results with nine real-world rootkits show that HookSafe is effective in defeating their hook-hijacking attempts. Our performance benchmarks show that HookSafe only adds about 6% performance overhead.

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 US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, and 0910767. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] GCC Extension for Protecting Applications from Stack-Smashing Attacks. *http://www.trl.ibm.com/projects/security/ssp/*.

[2] Stack Shield. *http://www.angelfire.com/sk/stackshield/info.html*.

[3] Trusted Boot. *http://tboot.sourceforge.net*.

[4] Advanced Micro Devices. *AMD64 Architecture Programmerś Manual Volume 2: System Programming*, 2007.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[6] ApacheBench - Apache HTTP Server Benchmarking Tool. *http://httpd.apache.org/docs/2.2/programs/ab.html*.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, , and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *Proceedings of the 7th USENIX Security Symposium*, 1998.

[10] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS '03: Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[11] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Security '09: Proceedings of the 18th USENIX Security Symposium*, 2009.

[12] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[13] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-based "Out-Of-the-Box" Semantic View Reconstruction. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[14] I. T. Lab. Attacking SMM Memory via Intel CPU Cache Poisoning. *http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf*.

[15] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *NDSS '09: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2009.

[16] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Security '08: Proceedings of the 17th USENIX Security Symposium*, 2008.

[17] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Oakland '07: Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.

[18] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Oakland '08: Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[19] N. L. Petroni, Jr. and T. Fraser. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Security '06: Proceedings of the 15th USENIX Security Symposium*, 2006.

[20] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Security '04: Proceedings of the 13th USENIX Security Symposium*, 2004.

[21] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[22] QEMU. *http://www.qemu.org*.

[23] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[24] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *EuroSys '09: Proceedings of the 4th European Conference on Computer Systems*, 2009.

[25] J. Rutkowska. System Virginity Verifier. *http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt*.

[26] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP '07: Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[27] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[28] A. Shevchenko. Rootkit Evolution. *http://www.viruslist.com/en/analysis?pubid=204792016*.

[29] UnixBench. *http://ftp.tux.org/pub/benchmarks/System/unixbench/*.

[30] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *DSN '05: Proceedings of the 35th International Conference on Dependable Systems and Networks*, 2005.

[31] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[32] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *NDSS '08: Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2008.

[33] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.