Transparent Protection of Commodity OS Kernels Using Hardware Virtualization

Michael Grace[1], Zhi Wang[1], Deepa Srinivasan[1], Jinku Li[1], Xuxian Jiang[1], Zhenkai Liang[2], and Siarhei Liakh[1]

[1] Department of Computer Science, North Carolina State University
[2] School of Computing,
National University of Singapore

Abstract. Kernel rootkits are among the most insidious threats to computer security today. By employing various code injection techniques, they are able to maintain an omnipotent presence in the compromised OS kernels. Existing preventive countermeasures typically employ virtualization technology as part of their solutions. However, they are still limited in either (1) requiring modifying the OS kernel source code for the protection or (2) leveraging software-based virtualization techniques such as binary translation with a high overhead to implement a Harvard architecture (which is robust to various code injection techniques used by kernel rootkits). In this paper, we introduce hvmHarvard, a hardware virtualization-based Harvard architecture that transparently protects commodity OS kernels from kernel rootkit attacks and significantly reduces the performance overhead. Our evaluation with a Xen-based prototype shows that it can transparently protect legacy OS kernels with rootkit resistance while introducing $< 5\%$ performance overhead.

Keywords: Virtualization, Harvard Architecture, Split Memory.

1 Introduction

Kernel rootkits are among the most insidious threats to computer security today. Embedding themselves within the operating system kernel, these rootkits enjoy unfettered access to the entire system and adopt various techniques to make themselves stealthy and "sticky," thus preventing them from being detected and removed. Given the effectiveness of this approach, it is not surprising that there has been explosive growth in the number of new rootkit families over recent years [2,4].

Kernel rootkit countermeasures have attracted a commensurate amount of attention in the research community. In particular, there are two main categories of existing efforts. The first category aims to detect the rootkit presence by looking for abnormalities or symptoms of rootkit infection. For example, Copilot [28] uses a special PCI card to grab a memory image of the kernel and then scans for any possible manipulation of kernel code or system-critical data structures. The follow-up efforts [29,30] extend it to detect any violation from semantic specifications of static and dynamic kernel data or

S. Jajodia and J. Zhou (Eds.): SecureComm 2010, LNICST 50, pp. 162–180, 2010.

deviation from the normal kernel control flow graph. However, these systems by design are all based on detecting rootkits *after* they have already installed themselves in the kernel.

In contrast, the second category strives to prevent rootkit infection in the first place by enforcing some security property. For example, SecVisor enforces a $W \oplus X$ property on kernel memory pages. The $W \oplus X$ property states that a given memory page can be either writable or executable, but not both at the same time. $W \oplus X$ enforcement is complicated by legacy OS kernels that contain mixed kernel pages with both code and data [20,21,22]. To handle such pages, SecVisor modifies the kernel source code to make the OS kernel memory layout conform to the $W \oplus X$ property. From another perspective, NICKLE [31] takes a software virtualization (i.e., binary translation) approach to emulate a Harvard architecture on x86, which essentially creates a separate memory space to reliably store authorized kernel code. By transparently redirecting kernel instruction fetches to the separate memory space, NICKLE is able to support unmodified kernels and guarantee their kernel code integrity, which effectively defeats most existing rootkits. However, from another perspective, the presence of a dedicated code memory and the need for transparent redirection of kernel instruction fetches require intercepting and redirecting every single kernel instruction execution, which unfortunately causes significant performance overhead [31].

In this paper, we introduce *hvmHarvard*, a hardware virtualization-based Harvard architecture on x86 that can not only transparently support commodity OSs without modification, but also effectively reduce the performance overhead. Specifically, we observe that the high performance overhead of implementing software-based Harvard architecture is mainly caused by *instruction-level* interception and redirection (of kernel instruction fetches) to the code memory. As such, we propose *page-level* redirection in hvmHarvard so that the performance overhead can be significantly reduced without unnecessarily sacrificing the security guarantee.

There are two main challenges involved in changing from instruction-level redirection to page-level redirection of kernel instruction fetches. The first one comes from the fact that x86 is not designed to support the Harvard architecture. To address that, we make an unconventional use of split code and data TLBs on x86 in combination with recent hardware-based tagged TLB support [3]. In particular, with separate code and data TLBs, we can dynamically adjust the page table to virtualize the Harvard architecture on top of x86 (so that code and data each have its own memory address space). The tagged TLB support is essential here as it avoids flushing the code/data TLBs in a virtualized environment (e.g., VM exits – Section 3.1), thus allowing the hypervisor to safely intervene and manipulate the guest page table in use for the Harvard architecture creation. With the separation of code memory and data memory, our Harvard architecture can naturally handle the mixed code and data pages in commodity OS kernels while still strictly enforcing $W \oplus X$. In the meantime, we also observe that the majority of existing kernel memory pages are not mixed. As a result, there is no need for hvmHarvard to keep a shadow copy of these pages, nor does it need to intervene on instruction fetches from them. By doing so, no processing overhead will be incurred on these pages and no extra memory space will be wasted, as they no longer need to be shadowed [31].

The second challenge stems from the need to perform *mode-sensitive* page-level redirection since we are interested in redirecting kernel instruction fetches only. In other words, we need to first determine the current running mode and then decide whether the corresponding instruction fetch should be redirected or not. This imposes a strict requirement to intercept every mode-switching event (e.g., including system calls) in the redirection logic. Intercepting these events at the hypervisor will cause significant performance overhead. Our solution to this problem involves altering the guest's view of memory at each privilege level (or mode): all of user memory becomes non-executable when a process is executing at the kernel mode, and vice versa. For brevity, we call this a mode-sensitive view (Section 3.2). During the normal operation of the guest, hvmHarvard does not intercept and mediate the change between different views of memory. Instead, our system injects trampoline code to switch between these two views of memory upon the mode-switching event inside the guest. The trampoline mechanism leverages an Intel hardware virtualization extension called the `CR3` Target Value List (Section 3.2) to avoid being trapped by the hypervisor and to achieve better performance.

We have implemented a Xen[9]-based proof-of-concept prototype. The prototype can transparently support a number of commodity systems including legacy Red Hat 8.0 (with a Linux 2.4.18 kernel) and recent Ubuntu 9.04 (running Linux 2.6.30-5). Our evaluation shows that our system is effective in preventing eight kernel attacks (including six real-world rootkits and two synthetic attacks) against legacy OS kernels that do not have the $W \oplus X$ support. Such protection is achieved with only a small performance overhead (i.e., $< 5\%$). To summarize, our paper has the following contributions:

- We propose a hardware virtualization-based Harvard architecture to effectively protect commodity OS kernels from kernel rootkit attacks. Compared with existing approaches, our system can not only achieve a similar protection guarantee, but also significantly reduce the performance overhead suffered by previous approaches.
- The first key technique in our approach is *page-level* redirection of instruction fetches, which departs from prior efforts that perform instruction-level redirection. Our technique significantly reduces the performance overhead in the creation of the Harvard architecture on top of x86.
- The second key technique enables *mode-sensitive* redirection by redirecting *only* kernel instruction fetches. In this way, we can effectively avoid hypervisor intervention in the guest's mode-switching events. As these events occur frequently inside the guest, this technique also contributes to reducing the overall performance overhead.
- Finally, we present a Xen-based system prototype. The evaluation results with the prototype confirmed the practicality and effectiveness of our approach.

The rest of the paper is structured as follows. We briefly describe necessary background on the Harvard architecture and hardware virtualization in Section 2. Our system design and implementation are then presented in Section 3 and Section 4, respectively. After that, we present the evaluation results in Section 5, which is followed by the discussion on possible limitations and their improvement in Section 6. Finally, we discuss related work in Section 7 and conclude our paper in Section 8.

2 Background

In this section, we briefly review some key concepts that are essential to our system but may be unfamiliar to some readers: the Harvard architecture and shadow paging in virtualization. Readers with sufficient background can safely skip this section.

2.1 Harvard Architecture

Modern computers use a single address space to refer to working memory. This model of memory is commonly known as the von Neumann architecture. Interestingly, some of the very earliest computers used two utterly separate working memories, one for instructions and one for data. This arrangement is known as the Harvard architecture (Figure 1). In a pure Harvard architecture machine, data accesses and instruction accesses are treated as accessing totally distinct address spaces. From a security standpoint, this addressing scheme eliminates code injection attacks. For example, some buffer overflow attacks use an overlong memory copy operation to overwrite memory that will be executed as code. A pure Harvard architecture machine is not vulnerable to this class of attacks, as their addressing scheme does not allow code and data to be referred to interchangeably.

This work focuses on a widely deployed processor family, *x86*, which has a unified address space for main memory and is thus a von Neumann architecture. However, x86 processors typically have separate caches for instructions and data. When executing from cache, the processor behaves like a Harvard architecture machine[1]. Only when main memory must be consulted, does x86 look like a von Neumann architecture. This observation is the foundation of our page-level redirection technique for the creation of the Harvard architecture on top of x86 (Section 3.1).

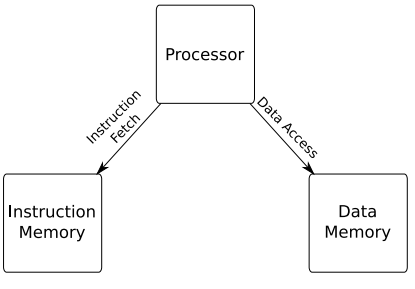

Fig. 1. The Harvard architecture

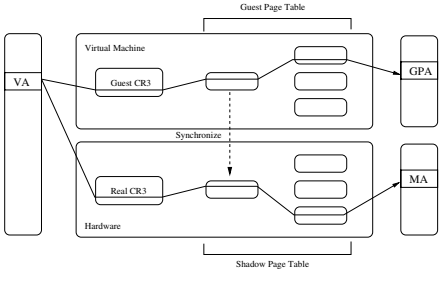

Fig. 2. Guest page table vs. shadow page table

2.2 Virtualization and Shadow Paging

Virtualization involves running a guest operating system in an environment that provides the illusion of complete access to a physical machine. All the resources used to construct such an illusory machine constitute a Virtual Machine (VM), while the

[1] This hybrid architecture is known as a *modified* Harvard architecture; many processors with the caching feature use such an arrangement today.

software that maintains one or more VMs is known variously as a hypervisor or a Virtual Machine Monitor (VMM). The hypervisor is commonly considered to be part of Trusted Computing Base (TCB) as it is strictly isolated from the VMs it manages and is often much smaller than modern operating systems.

There are several ways to virtualize a guest operating system. Since our work is based on hardware virtualization, we focus on its operation here. In particular, based on certain processor extensions, hardware virtualization operates a "trap-and-emulate" model. When a guest OS wishes to perform a privileged operation, the hardware has two options: either it can handle the request based on the processor extension for hardware virtualization, or if that is not possible, it can pass control to the hypervisor for handling. Handling the latter case constitutes a goodly portion of the hypervisor's workload and is typically an involved process.

Shadow paging is one such example. To better describe it, we first review how memory management works on an un-virtualized machine. Recall that x86 supports two memory protection mechanisms: segmentation and paging. They protect memory in a similar way by essentially permitting a higher-privilege piece of software to put blinders on a lower-privilege program, thus restricting its view of memory to only those things it is supposed to be able to access. Since segmentation support is being phased out in the new 64-bit long mode, we focus on the paging protection mechanism. In essence, paging uses page translation tables, or page tables for short, to remap memory for a given process. Virtual addresses are translated into physical addresses by these tables. These tables are also used by the hardware to enforce certain permissions policies (e.g., NX [1]) on the types of accesses allowed.

Virtualization has not changed this picture of the process; it has merely added another layer underneath it. By leveraging paging, the hypervisor divides the machine's memory into distinct logical machine memories. The guest OS in a VM then treats the memory it is given in the traditional way, dividing it up between the applications running in the guest. Under hardware virtualization, however, the OS itself does not know the *real* machine addresses that make up its allotted memory. With shadow paging, the hypervisor solves this problem by introducing an extra layer of indirection. In particular, a shadow table is created for a guest and maintained in the hypervisor. An unsuspecting guest OS kernel is allowed to maintain its own page tables, but they are not actually used by the hardware. Instead, the hypervisor marks these guest page tables read-only. Any attempt to write to them therefore generates a page fault, which is trapped by the hypervisor. The hypervisor, in turn, emulates the write request, eventually outputting the equivalent entry into the "real" page table used by the hardware. The guest can never see this real page table, which is assiduously kept synchronized with the one it can see – thus the name "shadow page table."

This arrangement is illustrated graphically in Figure 2. In the diagram, a virtual address (VA) is translated through both the guest's and the hardware's page tables. The guest's page tables eventually lead to a guest physical address (GPA) – the address the guest thinks of as being a hardware address. The shadow page tables instead translate the same virtual address into the real machine address (MA). The tables are kept synchronized by the hypervisor; this synchronization is represented by the dotted lines in the figure.

3 Design

In this work, we aim to develop a hardware virtualization-based Harvard architecture that can efficiently support unmodified legacy OS kernels and protect them from kernel rootkit attacks. Specifically, the presence of two distinct memory spaces for code and data in a Harvard architecture is useful for blocking code injection attacks and enforcing the $W \oplus X$ property. In this work, we propose to take a step further by enforcing mode-sensitive $W \oplus X$, also known as $W \oplus KX$. Due to our focus on OS kernel protection, $W \oplus KX$ requires that a user-level memory page will not be executable from the kernel mode and vice versa. Commodity hardware by default allows the execution of user-level memory pages at kernel privilege, which opens up "interesting" opportunities for kernel rootkit infection. As our defense, $W \oplus KX$ is proposed to effectively block this infection vector.

Threat Model and System Assumption. In this paper, we assume an adversary model where attackers or kernel rootkits are able to exploit software vulnerabilities in an OS kernel to launch code injection attacks. Accordingly, we also assume kernel rootkits have the highest privilege level inside the victim VM (e.g., the *root* privilege in a UNIX system) and have full access to the VM's memory space (e.g., through `/dev/mem` in Linux). However, the goal of a kernel rootkit is to stealthily maintain and hide its presence in the victim system; to do so, it 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 6.

In the meantime, our system assumes a trustworthy hypervisor as the necessary trusted computing base (TCB) to provide strict VM isolation. This assumption is shared by many other hypervisor-based security research efforts [13,14,17,25,43] and being hardened by existing hypervisor-protection solutions [26,41]. We will discuss possible attacks (e.g., VM escape) in Section 6. With this assumption, we consider the threat from layer-below attacks launched from physical hosts outside of the scope of this work.[2]

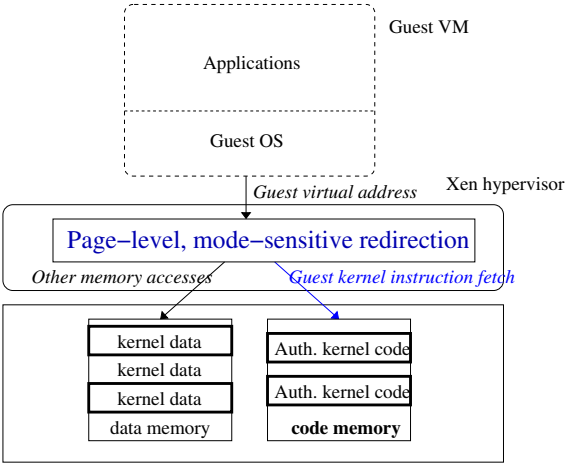

Fig. 3. Page-level mode-sensitive redirection enables an efficient implementation of the Harvard architecture on top of x86

[2] There exists another type of layer-below or specifically hardware DMA attack that is initiated from within a guest VM. However, since the hypervisor itself virtualizes or mediates guest DMA operations, recent hardware support for IOMMU can be readily adopted to intercede and block them. Therefore, we do not consider them in this paper.

3.1 Page-Level Redirection for $W \oplus X$

The central scheme of our approach is to efficiently create a Harvard architecture (Figure 3) on x86 by virtualizing one memory space for code and another for data. To achieve our goal, we observe the presence of separate TLBs for instruction fetches and data accesses. Note that each TLB entry caches the translation result from a virtual address to a physical address. When a memory access or an instruction fetch occurs, the virtual address lookup will go through the corresponding TLB first. Should that TLB not contain an entry for the requested translation (called a TLB miss), the hardware walks through the page table entries in main memory to do the lookup, then constructs such an entry. As a result, from the TLB's perspective, the hardware itself thinks in terms of two address spaces. However, in normal operation, these address spaces are kept synchronized and thus describe a unified memory space. Fortunately, to our benefit, there is no hardware requirement that this must be the case. In other words, to emulate a pure Harvard architecture, we can take advantage of these two TLBs by desynchronizing and loading them with two different page table entries for the same virtual address, thus creating two distinct memory spaces for code and data.

Unfortunately, the de-synchronization of these two TLBs is a delicate process, which is complicated by the fact that a TLB entry has a relatively limited lifespan. First, the TLBs are not large enough to cache all translation results at the same time, which means that older entries are eventually overwritten by newly-requested translations. Second, when an OS kernel either alters a page table or switches address contexts, these caches are implicitly flushed. Third, x86 provides very few instructions for interacting with the TLBs. In fact, after enabling the paging mode, the provided instructions are mainly used for removing one or all entries from *both* TLBs, which means the only way for us to populate a TLB entry will be by performing an address translation that eventually winds up in that cache.

To deal with the above challenges, we need to effectively intercept the hardware's attempts to re-populate TLBs. In particular, for the virtual addresses of interest, when there is a TLB miss, the hardware consults the page table and checks the permission bits of the entry it loads. If those permissions are violated, a page fault (or `#PF`) exception will be thrown. When there is a TLB hit, the cached entry's permissions are directly checked without consulting the page table. As a result, in the case of a TLB miss, we need to carefully prepare the page table in a way that will load the desired translation results as well as related permissions into respective TLBs.

There are three permission bits that can cause useful faults: the `USER` bit, the `PRESENT` bit and the `NX` bit. The `USER` bit only faults when a user-mode instruction fetch references a kernel page. With our focus on kernel protection, we are not interested in using this bit. The `PRESENT` bit, if not set, traps *any* access – which would lead to many expensive world switches. The `NX` bit causes a fault on any instruction fetch from pages with this bit set. In our system, we naturally leverage the `NX` bit.

In particular, to use the `NX` bit to cause one virtual address to map to two context-sensitive memory pages, we map the address to its data memory page and set its `NX` bit. If execution branches to an address within the page, the page fault handler substitutes its entry to code memory page and clears the `NX` bit. In order to load the entry into the instruction TLB (ITLB), the page fault handler must allow the guest to execute an

Algorithm 1. TLB de-synchronization algorithm

Input: Redirected Page Address (addr), Page table Entry for addr (pte)

```
/* handling NX-based page fault */ ;
    pte = the_code_page (addr);
    set_trap_flag ();
    return_to_guest ();

/* handling TF-based fault */ ;
    pte = the_data_page (addr) ;
    unset_trap_flag ();
    return_to_guest ();
```

instruction using this entry. However, once the code page entry has been loaded, the system needs to regain control to restore the map back to the data memory page. If this is not done, the data TLB (DTLB) may wind up being populated with the code page entry, routing data reads to the code page and thus violating the Harvard architecture. Note that the code page entry is marked as *read-only* and there is no way to cause a page to be executable yet not readable on the *x86* architecture.

To ensure that the page table is restored to the corresponding data entry as soon as possible, our design relies on the x86 single-step execution feature. Specifically, by setting the *trap* flag (or `TF`) of the `EFLAGS` register, the processor will generate an exception after every instruction. This feature allows us to execute one instruction, and then restore the data page entry in the `TF` handler. The process is shown in pseudo-code in Algorithm 1.

In this way, our design can populate the ITLB with one record and DTLB with another record without interfering each other. Here, we point out that if by trapping the execution of a guest VM to the hypervisor, a VM exit (or `VMEXIT`) occurs. In some processors, VM exits will flush the TLBs, which defeat our purpose of de-synchronizing TLBs. In our prototype, we leverage a hardware feature called tagged TLB [3] that is available in all recent hardware-virtualized AMD processors as well as Intel processors based on the new Nehalem architecture. This hardware feature essentially adds an extra field or an identification "tag" to each TLB entry that specifies the VM context within which the entry is valid. When a VM exit occurs, these entries will not be flushed. More details about our system will be presented in Section 4.1.

3.2 Mode-Sensitivity for $W \oplus KX$

By effectively creating a Harvard architecture on x86, our page-level redirection technique is able to enforce $W \oplus X$ while accommodating mixed kernel pages in commodity OS kernels. However, the $W \oplus X$ enforcement is still insufficient due to the need to block the execution of user-level pages from the kernel level. In other words, we need to enforce a stronger $W \oplus KX$ policy. As mentioned earlier, this is necessary as commodity OS kernels disallow the access of kernel memory pages from user mode, but do permit the execution of user memory pages from kernel mode.

To elaborate on this, the *x86* architecture has two related concepts in this vein: the `USER` page table permission bit and the Current Privilege Level (CPL) bits in the `CS` register. The CPL simply determines what instructions are valid – including access right checking on instruction fetches. The most-privileged CPL (or ring 0 where the kernel runs) has all the capabilities of the least-privileged CPL (or ring 3 where user-level applications run). Therefore, while it is illegal for a program executing at the ring 3

privilege to access kernel space, it is perfectly acceptable for a ring-0 kernel to branch its execution to user space.

With $W \oplus KX$, we aim to define a new Kernel eXecute (KX) mode of operation. In this mode, instruction fetches only succeed if the privilege level of the machine matches the privilege level of the page table entry. In other words, if `USER` is cleared for a page table entry, it is only executable at CPL=0, and when `USER` is set, it is only executable at CPL=3.

To achieve this, we propose maintaining *two* shadow page tables instead of one in the normal situation: one for user-privilege (or mode) execution and one for kernel-privilege execution. Each has the `NX` bit set for the opposite privilege's pages. A straightforward approach would require the hypervisor to intervene and swap the shadow page table upon every mode switch, from user to kernel and vice versa. Unfortunately, this scheme would induce a large number of costly `VMEXIT`s – two for every system call. To reduce this overhead, note that modern processors introduce special instructions – `sysenter`/`sysexit` to enable fast transfers between user and kernel. As these instructions use registers to point to the entry point of the system call handler, by redirecting that register to our trampoline code, we can handle a large number of mode switches in a performance-efficient fashion. More specifically, our approach leverages a hardware feature known as the "`CR3` Target Value List."[5] This feature is designed to allow a hypervisor to whitelist a set of expected `CR3` values: when a guest changes `CR3` to one of these values, the hypervisor is not consulted, saving a significant number of cycles that would be wasted on a world switch. In our prototype, our system injects a trampoline into the guest that simply switches page tables upon each mode switch, before the actual OS system call handler is invoked. Similarly, we use this trampoline to switch the page tables again before the system call handler returns back to user mode.

We assert that this optimization does not harm the $W \oplus KX$ security guarantee offered by our system. Specifically, the trampoline code is located on a page that the hypervisor prevents the guest from modifying. Also, if the guest invokes the trampoline code in an unintended way, it will always wind up either transferring control to the `sysenter`/`syscall` handler or executing the corresponding return instruction. From the OS kernel's perspective, the $W \oplus X$ property is not violated. More detailed discussion will be presented in Section 6.

Finally, it is worth mentioning that our system follows the same steps proposed in NICKLE to support loadable kernel modules (LKMs) [31]. In particular, we simply verify the hash signature of such drivers (and the main kernel) when they are being loaded. For example, for Linux kernels, we leverage the fact that the kernel's module loader calls the `init()` method of a module when it is being loaded. As this will cause a page fault due to our page-redirection technique, we can check the instruction pointer (`IP` register) to see if it matches an address within the kernel's module loader. If it does, the system can locate the module definition structure and use that information to determine how to verify the module. Falsifying the module structure information would inevitably result in a hash signature inconsistent with the trusted version of the module, causing the falsified module to be simply rejected by our system. Note that we do not need to modify the guest operating system; our system simply needs to know how to find the information it needs in the guest operating system's memory. Such knowledge

can be provided in a number of ways, e.g., either directly compiled into the hypervisor, loaded in the VM's metadata or indirectly hinted to the hypervisor from a hypercall within the VM.

4 Implementation

We have developed a proof-of-concept prototype on top of Xen 3.3.1, targeting fully-virtualized 32-bit legacy guests running under a 32-bit PAE hypervisor. Our development was tested against a Red Hat 8.0 image (running a Linux 2.4.18 kernel) and an Ubuntu 9.04 image (running a Linux 2.6.30-5 kernel). Our development machine had a Core i7-930 Nehalem processor with recent hardware virtualization support. Our current prototype only supports a single virtual CPU for one guest and the support of SMPs are left to future work. In the following, we present additional implementation details for the two key techniques in our approach.

4.1 Page-Level Redirection

As mentioned earlier, our scheme virtualizes a pure Harvard architecture machine on x86 by using a hypervisor to desynchronize the processor's TLBs. Naturally, our prototype mainly deals with various particulars of the x86 paging mechanism and related TLB operations. In particular, our experience indicates that there is a strong correlation between the frequency with which the TLBs must be fixed up and the performance overhead of the system as a whole. Note the process of de-synchronizing or splitting a page's TLB entries is a costly operation. Each time a page needs to be split, there are two associated `VMEXIT`s: one caused by the NX-based page fault to populate the ITLB, and another from the single step fault handler to populate the DTLB. Because of that, it is critical to avoid generating these events if possible.

In our prototype, we implement an optimization that is akin to the traditional copy-on-write (COW) technique. Recall that one main purpose of our system is to ensure $W \oplus X$. As such, if some kernel pages in commodity OSs are already amenable for $W \oplus X$ enforcement, we can simply enforce it without needing to create two separate copies (one for code and one for data) in the first place. By doing so, we can not only avoid allocating additional memory spaces in storing copies, but also reduce the number of `VMEXIT`s that would otherwise be needed to maintain the separate presence of code and data copies.

To further elaborate that, consider the impact of splitting a kernel page[3]. If the kernel page is never used as code, the additional overhead will be incurred when generating and maintaining the two copies, though there is little or no performance impact. However, if the kernel page is never used as data, then we will be splitting the page every time it is executed and the translation is not cached in the ITLB (or already flushed from the ITLB). As mentioned earlier, this process will involve the hypervisor and cause `VMEXIT`s, resulting in a high performance overhead.

[3] Xen's concept of kernel pages can be different than the guest OS'. For example, Xen does not internally use 2M or 4M "superpages"; if the guest OS allocates these, Xen treats them as a large number of normal 4K pages.

In our prototype, to determine the liveness of a kernel page, we perform basic reference-counting and dynamically track the number of times a given kernel page is referenced by the guest's page tables. In addition, by counting the number of writable mappings to a given kernel page, our system can intelligently choose *not* to split the page if that count is zero. In this way, we can further avoid unnecessary `VMEXIT`s for better performance.

4.2 Mode-Sensitivity Support

To make the page-level redirection mode-sensitive, we implement two shadow pages tables: one for guest user-mode and another for guest kernel-mode. As a result, every time the guest OS wishes to make a change to its page tables, the hypervisor intercepts the change and synchronizes it with the two shadow pages. As synchronization will require the hypervisor to walk through the shadow page tables and make the corresponding hardware-visible change, the presence of two shadow page tables will double the cost of synchronization. To reduce the cost, our prototype opts to interleave two page tables; this allows a single walk through them to find both entries related to a particular page table update. Specifically, for each page table bifurcated in this way, twice the normal amount of memory for shadow page tables is allocated. The low-order version of the page table is used for the guest kernel mode, and the high-order version is for the guest user mode. With that, one walk is needed to find the location to alter, followed by a privilege-level check that determines which changes to make and where to look for the second copy of that page.

With the two shadow page tables in place, our prototype further takes another optimization. Considering the fact that page tables are laid out in a layered hierarchy, we can trade granularity for ease of updating, simply by having two distinct top-level page tables map down to the same set of level-1 page tables (see Figure 4). The top levels of the page table are not altered as frequently as the lower levels are, leading to disproportionately less update overhead. They are also smaller (as there are fewer such top-level entries), leading to less cache pressure when compared to the case where all entries had to be maintained separately. Using a 32-bit Linux guest as an example, the Linux kernel occupies the top one gigabyte of address space. As the shadow page tables are 32-bit PAE tables, this neatly corresponds to one of the four top-level entries. Though the top-level entries do not have the `NX` permission bit, we can maintain two sets of the level-2 page tables instead that have the `NX` permission bit.

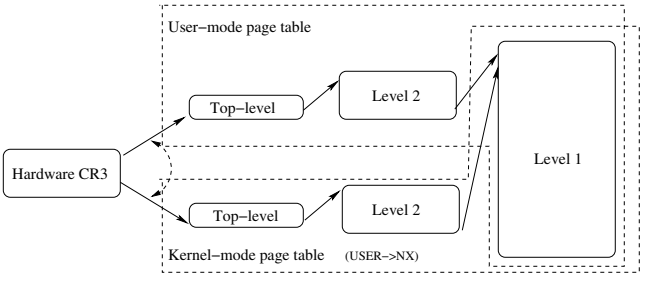

Fig. 4. Two shadow page tables: the user-mode page table and the kernel-mode page table share the same level-1 entries, but *not* top-level and level-2 entries

Table 1. Effectiveness of our system

Rootkit	Attack Vector	Prevented?	Result
adore-ng 0.56	LKM	Yes	Module fails to load
superkit	`/dev/kmem`	Yes	Crashes
mood-nt 2.3	`/dev/kmem`	Yes	Crashes
sk2rc2	`/dev/kmem`	Yes	Crashes
eNYeLKM 1.2	LKM	Yes	Module fails to load
Phalanx b6	`/dev/mem`	Yes	Crashes
synthetic-1	LKM	Yes	Module fails to modify itself
synthetic-2	LKM	Yes	`insmod` crashes

Afterwards, the two shadow pages will be switched based on the current running mode of the guest VM. In our prototype, we hook the handler for the `sysenter` instruction (by detouring the corresponding Model Specific Register or MSR content) to capture the user-to-kernel mode switch. Similarly, we also detour the `sysexit` execution by performing a kernel-to-user switch. We point out that such detouring happens inside the guest context with a trampoline without involving the hypervisor, thus avoiding unnecessary `VMEXIT`s. However, from another perspective, our prototype can still function properly without hijacking them because the hypervisor will simply step in and switch page tables itself, though at a lower pace.

An astute reader may observe that the trampoline code will essentially change `CR3`, the page table base address register. Changes to `CR3` will typically be trapped by the hypervisor. Fortunately, a recent hardware feature, i.e., the `CR3` Target Value List, allows our page table switch without being trapped by the hypervisor if the new `CR3` value is on the target value list. However, the `CR3` update is still considered a context switch, which unfortunately causes an unnecessary TLB flush – purging any split entries from the instruction TLB. Interestingly, the related level-1 page table entries contain a `GLOBAL` bit that can prevent a TLB flush from purging a particular entry.

There is a subtle issue in the interplay between the `CR3` Target Value List and the `GLOBAL` bit. By definition, the hypervisor is not alerted if `CR3` is changed to a value on the list. Likewise, if a split entry in the TLB is not purged, the page tables will not be consulted upon an instruction fetch to its virtual address. Therefore, if our user-mode `CR3` value is loaded from a page that is marked `GLOBAL`, execution could branch to user land while still at high privilege! Fortunately, there are only two ways that `CR3` can take a new value: via hardware task switching (`ltr`) or through the explicit assignment (`mov cr3, <general register>`). Hardware task switching is not used by either Windows or Linux.[4] For the more common `mov cr3` operation, we ensure that the instruction pointer, after a `mov cr3, <register>` operation, will always point to a virtual address that does not map to a TLB entry with the `GLOBAL` bit set. To assure that, we can scan each page as it is being split, ensuring that the opcode for this dangerous operation does not occur. In other words, we look for that string of bytes

[4] Note that even if it is used, the `ltr` operation acts on tables that are privileged and hardware virtualization allows for trapping the `ltr` operation. In other words, we can still prevent hardware task switching from breaking our $W \oplus KX$ guarantee.

throughout the split page. If it is found, the split code will ensure that upon every insertion to the ITLB, that split page's entry will not have the GLOBAL permission bit set.[5]

5 Evaluation

To test the effectiveness of our prototype, we run six real-world rootkits and two synthetic exploits (both violate $W \oplus KX$) against a default Ubuntu 9.0.4 system. These attacks were selected as representative of the infection vectors used by existing kernel rootkits. In every case, our system was able to defeat the infection and protect the system. In the following, we present details of two representative experiments.

Mood-NT Rootkit Experiment. Some rootkits install themselves by directly writing to mixed pages in kernel memory. In this experiment, the *mood-nt* rootkit [31] uses the /dev/kmem interface to access kernel memory through the file system. Specifically, the rootkit uses the interface to copy its resident logic into kernel memory, and then overwrites function pointers to hijack the kernel's control flow.

When the test system is protected under our prototype, code injection appears to work fine as the injected content is directly written into the data page. However, when one of the rootkit's function pointers is called, our page-level redirection technique immediately causes the resulting instruction fetch to a code page, *not* the data page that contained the injected content. As a result, instead of fetching the rootkit's code, the processor attempts to execute whatever is in the code page, eventually leading to a crash in our experiment.

Synthetic Attacks. In this experiment, we intentionally play with the $W \oplus KX$ protection by redirecting kernel control flow to user-space code. Since we do not have a rootkit sample that was developed in this way, we simply synthesize an attack that would execute user code at kernel privilege.

Specifically, we implemented a branch-to-userspace exploit as a loadable kernel module. In the module's initialization function, we create a pointer to an address within insmod's address space. This address in user space contains an instruction sequence that copies the top of the stack into EBX and then returns. Therefore, after successfully executing it, EBX should equate to EIP. Running under hvmHarvard, the execution faults to the hypervisor when the first user instruction is fetched. From the page fault handler, it reports the fault as a NX violation and relays it to the guest OS kernel, which then terminates the insmod process.

Performance Overhead. To evaluate the impact on system performance, we have performed benchmark-based measurements. In particular, we use two application-level benchmarks and one microbenchmark to evaluate the system. They are (1) a normal

[5] Note that there are a few corner cases worth mentioning. The mov cr3, <register> instruction is translated to 0f 22 d? in machine code. If the split page ends neatly with 0f 22 d?, then it would put the instruction pointer onto the next page, whose GLOBAL property is uncertain. Fortunately, that case does not occur in the Linux kernels we have examined. Such a special case can also be handled upon insertion into the TLB, by proactively re-populating the next page's TLB entry as ¬GLOBAL.

Table 2. Software configuration for performance evaluation

Item	Version	Configuration
Ubuntu	9.0.4	Using Linux 2.6.30
Apache	2.0.59	Using the default high-performance configuration file
Kernel	2.6.30	Standard kernel compilation
ApacheBench	2.0.40-dev	`ab -c3 -t 60 <url/file>`
LMbench	3.0alpha	Using the default configuration

compilation of the Linux 2.6.30 kernel, (2) network throughput test on the Apache web server using the ApacheBench [8], and (3) a standard system benchmark toolkit called LMbench [24]. Our tests were performed on a Dell Optiplex, which runs the Ubuntu 8.04 system and has an Intel Core i7-920 (2.66GHz) CPU and 4GB RAM. The guest VM runs Ubuntu 9.04 with Linux kernel 2.6.30-5 and 1GB of memory. For comparison, we run the guest VM on Xen 3.3.1 twice, with and without protection. The software configuration for our evaluation is shown in Table 2. The benchmark programs were run ten times and averaged. Our results are shown in Table 3.

Table 3. Application benchmark results. For `make`, lower is better; for Apache, higher is better.

Benchmark	Without protection	with protection	Overhead
`make kernel`	41.289 s	43.312 s	4.9%
ApacheBench	11728.68 req/s	11497.24 req/s	2.0%

In our first application benchmark, we compiled our guest VM's kernel with the command `'make kernel'`, using `time` to measure how long the process took. The system under protection takes 44.275 seconds to complete, which is 4.9% longer than the compilation time in an unprotected system. In our next application benchmark, we set up an Apache [7] web server. The ApacheBench program, `ab`, was run against a small (15K) html file on that server. We then collected the network throughput and the results show a 2.0% slowdown. We also evaluated our system with LMbench [24], which is a micro-benchmark for OS kernel performance. The tasks include process creation, basic arithmetic operations, context switching, file system operation, local communication, and memory latency. Among these results, the maximum overhead of our system is 4.70% when doing context switching. The overhead comes from updating the `CR3` Target Value List that is used for later switching of the two shadow page tables. Other tasks such as performing basic arithmetic or floating-point operations incur the lowest overhead, which is nearly zero.

6 Discussion

In this section, we discuss several issues related to our system. First, our goal here is to efficiently create a Harvard architecture on x86 and enable $W \oplus KX$ for kernel code integrity protection. As a result, our system is not able to protect the kernel control-flow integrity. In other words, an attacker could possibly launch a "return-into-libc" style attack or the so-called return-oriented attack [10,16,37] within the kernel by leveraging

only the existing authenticated kernel code. Fortunately, solutions exist for protecting control flows [6,15,30,42] and data flow integrity [11] for user-level applications, which could be potentially extended to complement our system for kernel protection.

Second, as with existing systems for kernel code integrity, our current implementation does not support self-modifying kernel code. This limitation can be removed by intercepting the self-modifying behavior (e.g., by trapping and validating the self-modification behavior) and re-authenticating and updating the kernel code in the code memory after the modification.

Third, our system currently does not support kernel page swapping. Linux does not swap out kernel pages, but Windows does have this capability when under heavy memory pressure. Supporting kernel page swapping would require intercepting swap-out and swap-in events and ensuring that the page being swapped in has not been maliciously tampered with.

Fourth, hvmHarvard cannot take advantage of the hardware-assisted paging mechanisms built into modern AMD and Intel processors [3,5]. These schemes do not require the hypervisor to intervene when the guest wishes to alter its page table (as in shadow paging), resulting in superior performance. Unfortunately, our page-level redirection scheme requires page table updates be registered with the hypervisor. Consequently, further work would be required to adapt our scheme to use hardware-assisted paging.

Finally, we point out that our scheme assumes a trustworthy hypervisor to enforce $W \oplus KX$. This assumption is needed because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. We also acknowledge that a VM environment can potentially be fingerprinted and exploited [18,33] by attackers. Fortunately, recent solutions on hypervisor protection [19,23,41] can be employed to thwart these attacks. Also notice that as virtualization continues to gain popularity, the concern over VM detection may become less significant as attackers' incentive and motivation to target VMs increase.

7 Related Work

Kernel Rootkit Detection. A number of systems have been proposed to detect the presence of kernel rootkits. Some of them passively validate kernel code and examine kernel data for signs of infection. For example, System Virginity Verifier [34] validates the integrity of the Windows instance that it runs within. As running inside a compromised operating system is dangerous, Copilot [28] copies operating system memory onto a PCI card for analysis by a dedicated co-processor. Further extensions allow it to detect breaches of kernel data semantic integrity [29] and state-based control flow integrity [30]. Strider GhostBuster [40] and VMwatcher [17] aim to look for discrepancies between an internal and external view of a system to detect the hiding behavior from rootkits.

Recently, Lares [27] and its in-VM equivalent, SIM [38], attempt to create secure kernel hooks that can be used to monitor system events. In particular, SIM is capable of installing hooks into a virtualized guest that run code safely *without* hypervisor intervention. SIM uses the same Intel `CR3` Target Value List feature that our work does, but uses it to create a safe introspection environment instead of a new paging feature as in our system.

Kernel Rootkit Prevention. Rather than detecting rootkits already resident in an OS kernel, other systems attempt to protect the kernel from being infected in the first place. Livewire [14] is among the first in using virtualization techniques for this purpose, though the system mainly focuses on the protection of static kernel code and data structures. SecVisor [36] is a small security hypervisor that aims to securely enforce a $W \oplus X$ guarantee over memory but it requires modifying the OS kernel for the support. In other words, it is not able to support legacy OSs such as Redhat 8.0. Also note that SecVisor implemented a similar KX paging mode, but its shadow page table implementation uses a single page table per process, which leads to considerable performance overhead [36]. Instead, our approach proposes two page tables. Further, with the `CR3` Target Value List hardware virtualization feature, our system allows a guest running under our system to switch between these two page tables without hypervisor intervention. In the same vein, NICKLE [31] aims to protect the integrity of the kernel code with a software-based implementation of the Harvard architecture. The software implementation is based on instruction-level redirection, which has a high performance overhead. In comparison, our approach proposes a page-level, mode-sensitive redirection that substantially reduces the performance overhead.

More recently, Overshadow [12] is another related system. Its basic premise is that the kernel cannot be trusted with sensitive user data, even if it is not compromised or actively malicious. Like our system, Overshadow captures the mode-switching changes to alter the view of memory inside a protected VM. However, the differences are twofold: (1) First, our system switches between user and kernel page tables on each mode switch but do not attempt to encrypt user memory pages. In comparison, Overshadow makes the user memory appear encrypted to the operating system kernel, yet acts as normal when at user privilege; (2) Second, the goal of our system is to protect the kernel from malicious user applications while Overshadow does the exact reverse.

In addition to these techniques, there have been attempts to use lightweight virtual machines in place of processes. For example, the Qubes [35] operating system uses Xen to manage AppVMs each containing an application and a small Linux environment. AppVMs are treated analogously to processes, instead of as full-on virtual machines: functions such as storage and networking are handled centrally in dedicated, hardened virtual machines. While the isolation guarantees from such methods are potentially very strong, they are not a drop-in solution for legacy systems, due to their radically different interface.

TLB Manipulation. Finally, the presence of separate TLBs has been recognized and exploited in other contexts for different applications. For example, Wurster et al. [44] proposes using different ITLB and DTLB mappings to attack self-checksumming code. Almost simultaneously, Sparks and Butler [39] shows a rootkit prototype called Shadow Walker that could elude existing detection using the de-synchronized TLB. Later, Rosenblum et al. [32] demonstrates a system that used a modified version of Xen to instrument a tamper-resistant process within a VM. While the version of Xen used is unclear, it appears that their system operated on para-virtualized guests. In contrast, our system is mainly concerned with fully-virtualized guests and aims to defeat existing kernel rootkits. To the best of our knowledge, no other system has exploited recent hardware virtualization features to efficiently implement the Harvard architecture on x86, including the

use of tagged TLBs to manipulate the TLBs of a guest from outside as well as the unique hardware feature of the CR3 Target Value List.

8 Conclusion

In this paper, we present hvmHarvard, a hardware virtualization-based, efficient implementation of the Harvard architecture on top of x86. The Harvard architecture has two memory spaces (one for code and one for data) and is thus inherently robust to code injection attacks employed by most existing kernel rootkits. Different from prior efforts in using the instruction-level redirection to virtualize the Harvard architecture, our approach proposes a page-level, mode-sensitive scheme to achieve the same goal but with a significantly reduced performance overhead. We have implemented a Xen-based prototype. Our evaluation shows that it allows for transparent support of legacy OSs (without modification) as the guest and protects them from existing kernel rootkit attacks with a small performance overhead ($< 5\%$).

Acknowledgments. The authors would like to thank the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. 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 ARO and the NSF.

References

1. W^X, http://en.wikipedia.org/wiki/W_xor_X
2. Rootkit Numbers Rocketing UP, McAfee Says (2006), http://news.cnet.com/2100-7349_3-6061878.html
3. AMD Virtualization (AMD-V) Technology (2009), http://sites.amd.com/us/business/it-solutions/usage-models/virtualization/Pages/amd-v.aspx
4. Cooperation Grows in Fight Against Cybercrime (2010), http://www.avertlabs.com/research/blog/index.php/category/rootkits-and-stealth-malware/
5. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide (2010), http://www.intel.com/assets/pdf/manual/253669.pdf
6. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity Principles, Implementations, and Applications. ACM Transactions on Information and System Security 13(1), 1–40 (2009)
7. Apache Http Server Project, http://httpd.apache.org/
8. ab - Apache Benchmarking Tool, http://httpd.apache.org/docs/2.2/programs/ab.html
9. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: SOSP 2003: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)

10. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: CCS 2008: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 27–38. ACM, New York (2008)
11. Castro, M., Costa, M., Harris, T.: Securing Software by Enforcing Data-Flow Integrity. In: OSDI 2006: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 147–160. USENIX Association, Berkeley (2006)
12. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2–13. ACM, New York (2008)
13. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In: OSDI 2002: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, pp. 211–224. ACM, New York (2002)
14. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the Network and Distributed Systems Security Symposium, pp. 191–206 (2003)
15. Grizzard, J.B.: Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems. Ph.D. thesis, Georgia Institute of Technology (2006)
16. Hund, R., Holz, T., Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: Security 2009: Proceedings of the 18th USENIX Security Symposium (2009)
17. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-based "Out-of-the-Box" Semantic View Reconstruction. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 128–138. ACM, New York (2007)
18. Klein, T.: ScoopyNG (2010), http://www.trapkit.de/research/vmm/scoopyng/
19. Kortchinsky, K.: Honeypots: Counter Measures to VMware Fingerprinting (2004), http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html
20. Liakh, S., Jiang, X.: [2/4,tip:x86/mm] Set First MB as RW+NX (2010), https://patchwork.kernel.org/patch/90048/
21. Liakh, S., Jiang, X.: [3/4,tip:x86/mm] NX Protection for Kernel Data (2010), https://patchwork.kernel.org/patch/90046/
22. Liakh, S., Jiang, X.: [4/4,tip:x86/mm] RO/NX Protection for Loadable Kernel Modules (2010), https://patchwork.kernel.org/patch/90047/
23. Liston, T., Skoudis, E.: On the Cutting Edge: Thwarting Virtual Machine Detection (2006), http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf
24. LMbench - Tools for Performance Analysis (1998), http://www.bitmover.com/lmbench/
25. Lombardi, F., Di Pietro, R.: KvmSec: A Security Extension for Linux Kernel Virtual Machines. In: SAC 2009: Proceedings of the 2009 ACM Symposium on Applied Computing, New York, NY, pp. 2029–2034 (2009)
26. Murray, D.G., Milos, G., Hand, S.: Improving Xen Security through Disaggregation. In: VEE 2008: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 151–160. ACM, New York (2008)
27. Payne, B.D., Carbone, M., Sharif, M.I., Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization. In: Oakland 2008: IEEE Symposium on Security and Privacy (S&P 2008), pp. 233–247. IEEE Computer Society, Los Alamitos (2008)

28. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor. In: Security 2004: Proceedings of the 13th USENIX Security Symposium, pp. 179–194. USENIX Association, Berkeley (2004)
29. Petroni, Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Security 2006: Proceedings of the 15th USENIX Security Symposium, pp. 289–304. USENIX Association, Berkeley (2006)
30. Petroni, Jr., N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 103–115 (2007)
31. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
32. Rosenblum, N.E., Cooksey, G., Miller, B.P.: Virtual Machine-provided Context Sensitive Page Mappings. In: VEE 2008: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 81–90. ACM, New York (2008)
33. Rutkowska, J.: Red Pill (2004), `http://invisiblethings.org/papers/redpill.html`
34. Rutkowska, J.: System Virginity Verifier: Defining the Roadmap for Malware Detection on Windows System (2005), `http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt`
35. Rutkowska, J., Wojtczuk, R.: Qubes OS Architecture (2010), `http://qubes-os.org/`
36. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel code Integrity for Commodity OSes. In: SOSP 2007: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 335–350. ACM, New York (2007)
37. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561. ACM, New York (2007)
38. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure In-VM Monitoring Using Hardware Virtualization. In: CCS 2009: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 477–487. ACM, New York (2009)
39. Sparks, S., Butler, J.: Shadow Walker.: Raising the Bar for Rootkit Detection. In: Black Hat Japan (2005)
40. Wang, Y.M., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. In: DSN 2005: Proceedings of the 2005 International Conference on Dependable Systems and Networks, pp. 368–377. IEEE Computer Society, Los Alamitos (2005)
41. Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: Oakland 2010: IEEE Symposium on Security and Privacy (S&P 2010), pp. 380–398. IEEE Computer Society, Los Alamitos (2010)
42. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering Kernel Rootkits with Lightweight Hook Protection. In: CCS 2009: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 545–554. ACM, New York (2009)
43. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
44. Wurster, G., Oorschot, P.C.v., Somayaji, A.: A Generic Attack on Checksumming-Based Software Tamper Resistance. In: Oakland 2005: Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005), pp. 127–138. IEEE Computer Society, Los Alamitos (2005)