# SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes*

Arvind Seshadri
CyLab/CMU
Pittsburgh, PA, USA
arvinds@cs.cmu.edu

Mark Luk
CyLab/CMU
Pittsburgh, PA, USA
mluk@ece.cmu.edu

Ning Qu
CyLab/CMU
Pittsburgh, PA, USA
quning@cmu.edu

Adrian Perrig
CyLab/CMU
Pittsburgh, PA, USA
perrig@cmu.edu

## ABSTRACT

We propose SecVisor, a tiny hypervisor that ensures code integrity for commodity OS kernels. In particular, SecVisor ensures that only user-approved code can execute in kernel mode over the entire system lifetime. This protects the kernel against code injection attacks, such as kernel rootkits. SecVisor can achieve this property even against an attacker who controls everything but the CPU, the memory controller, and system memory chips. Further, SecVisor can even defend against attackers with knowledge of zero-day kernel exploits.

Our goal is to make SecVisor amenable to formal verification and manual audit, thereby making it possible to rule out known classes of vulnerabilities. To this end, SecVisor offers small code size and small external interface. We rely on memory virtualization to build SecVisor and implement two versions, one using software memory virtualization and the other using CPU-supported memory virtualization. The code sizes of the runtime portions of these versions are 1739 and 1112 lines, respectively. The size of the external interface for both versions of SecVisor is 2 hypercalls. It is easy to port OS kernels to SecVisor. We port the Linux kernel version 2.6.20 by adding 12 lines and deleting 81 lines, out of a total of approximately 4.3 million lines of code in the kernel.

**Categories and Subject Descriptors:** Software, Operating Systems, Security and Protection, Security Kernels.

**General Terms:** Security.

**Keywords:** Hypervisor, Code Attestation, Code Integrity, Preventing Code Injection Attacks, Memory Virtualization.

*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grant CCF-0424422 from the National Science Foundation, and equipment donations from AMD and KDDI. Arvind Seshadri's attendance at SOSP 2007 is supported by a student travel scholarship from the National Science Foundation. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AMD, ARO, CMU, KDDI, NSF, or the U.S. Government or any of its agencies.


*SOSP'07,* October 14–17, 2007, Stevenson, Washington, USA.


## 1. INTRODUCTION

Computing platforms are steadily increasing in complexity, incorporating an ever-growing range of hardware and supporting an ever-growing range of applications. Consequently, the complexity of OS kernels is steadily increasing. The increased complexity of OS kernels also increases the number of security vulnerabilities. The effect of these vulnerabilities is compounded by the fact that, despite many efforts to make kernels modular, most kernels in common use today are monolithic in their design. A compromise of any part of a monolithic kernel could compromise the entire kernel. Since the kernel occupies a privileged position in the software stack of a computer system, compromising it gives the attacker complete control of the system.

In view of the importance of the security of the kernel to the security of a system, securing existing kernels is of critical importance. Approaches that do not mandate large-scale design changes to existing kernels are preferable, since they will ease deployment. SecVisor represents a first step in that direction, as it provides a lifetime guarantee of the integrity of the code executing with kernel privilege. In other words, SecVisor prevents an attacker from either modifying existing code in a kernel or from executing injected code with kernel privilege, over the lifetime of the system. SecVisor can achieve this guarantee even in the presence of an attacker who controls everything on the system except for the CPU, memory controller, and system memory chips. SecVisor ensures that only code approved by the user can execute with kernel privilege. Users can supply their desired approval policy and SecVisor checks all code loaded into the kernel against the users' approval policy. It further ensures that the approved code currently in memory cannot be modified by the attacker.

SecVisor prevents numerous attacks against current kernels. For example, there are at least three ways in which an attacker can inject code into a kernel. First, the attacker can misuse the modularization support that is part of many current kernels. Modularization support allows privileged users to add code to the running kernel. An attacker can employ a privilege escalation attack to load a module into the kernel. Second, the attacker can locally or remotely exploit software vulnerabilities in the kernel code. For example, the attacker can inject code by overflowing a vulnerable kernel buffer. The NIST National Vulnerability Database shows that the Linux Kernel and Windows XP SP2 were found to have 81 and 31 such vulnerabilities in 2006. Third, DMA-capable peripheral devices can corrupt kernel memory via DMA writes. A sample attack that uses Firewire peripherals was demonstrated by Becher et al. [3].

We implement SecVisor as a tiny hypervisor that uses hardware memory protections to ensure kernel code integrity. SecVisor virtualizes the physical memory, which allows it to set hardware protections over kernel memory, that are independent of any protections set by the kernel. SecVisor uses the IO Memory Management Unit (IOMMU) to protect approved code from Direct Memory Access (DMA) writes. Also, SecVisor virtualizes the CPU's Memory Management Unit (MMU) and the IOMMU. This ensures that SecVisor can intercept and check all modifications to MMU and IOMMU state.

We have three design goals for SecVisor: (1) small code size to facilitate formal verification and manual audit, (2) limited external interface to reduce the attack surface, and (3) minimal kernel changes to facilitate porting commodity kernels. We have implemented SecVisor on a system with an AMD CPU, and ported the Linux kernel as our sample commodity kernel. To reduce the complexity of SecVisor, our implementation uses CPU-based virtualization (AMD's Secure Virtual Machine (SVM) technology [1]) to virtualize the physical memory, the MMU, and the IOMMU. Intel's Trusted eXecution Technology (TXT) (formerly called LaGrande Technology (LT)) [10] provides facilities similar to SVM. Using CPU-based virtualization support does not limit the deployment of SecVisor since such support is now widely available on both Intel and AMD CPUs. We use the Device Exclusion Vector (DEV) protections [1] of the SVM technology to protect kernel memory from DMA-writes by peripherals.

The rest of this paper is organized as follows. The next section discusses our assumptions and adversary model. Section 3 describes the design of SecVisor by stating the security properties required and describing how those properties can be achieved. Section 4 gives a brief background of the x86 architecture and SVM technology. Section 5 describes how we realize SecVisor using AMD's SVM technology. We show how we port the Linux kernel to SecVisor in Section 6 and present our evaluation of SecVisor in Section 7. Section 8 discusses the limitations and future implementation directions of SecVisor. Section 9 discusses related work and Section 10 concludes.

# 2. ASSUMPTIONS AND THREAT MODEL

In this section we state our assumptions and describe our threat model.

## 2.1 Assumptions

We assume that the CPU of the system on which SecVisor runs provides support for virtualization similar to AMD's SVM technology or Intel's Trusted eXecution Technology(TXT) [1, 10]. Also, we assume that the system has a single CPU and that the kernel protected by SecVisor does not use self-modifying code. In Section 8 we discuss how these two assumptions could be relaxed. On the x86 architecture, we assume that the kernel that executes in 32-bit mode. We also assume that SecVisor does not have any vulnerabilities. Given that the code size of SecVisor and its external interface are small, it could be possible to formally verify or manually audit SecVisor to rule out known classes of vulnerabilities.

## 2.2 Threat Model

We consider an attacker who controls everything in the system but the CPU, the memory controller, and system memory chips. This trusted computing base (TCB) is minimal for the *von Neumann architecture* (also called a *stored-program computer*) [23], which is the architecture used by most computing devices today. Examples attacks are: arbitrary modification of all memory contents, injection of malicious code into the system firmware (also called the BIOS on x86 systems), malicious DMA writes to memory using peripherals, and the use of malicious peripherals connected to the system. Also, the attacker might be aware of zero-day vulnerabilities in the kernel and application software on the system. The attacker may attempt to use these vulnerabilities to locally or remotely exploit the system. For the x86 architecture, we assume that the System Management Mode (SMM) [1] handler is not compromised by the attacker. In Section 8 we describe how this requirement can be relaxed.

# 3. SecVisor DESIGN

In this section, we discuss the design of SecVisor. We start off by describing the challenges involved. Then, we state the properties that need to be achieved in order to guarantee that only user-approved code can execute in kernel mode. Finally, we describe how SecVisor uses hardware memory protections combined with controlled entries and exits from kernel mode to achieve the required properties. This section presents the conceptual design of SecVisor that is independent of any CPU architecture or OS kernel. In Section 5 we describe how we realize an implementation of this conceptual design on the x86 architecture.

## 3.1 Challenges

We now discuss the challenges we face in designing an *enforcement agent* that enforces the guarantee of kernel code integrity over the lifetime of the system, under the assumption that our TCB consists of the CPU, the memory controller, and the system memory chips. The very first question we face is: where in the software stack of the system should the enforcement agent execute? The enforcement agent needs to be isolated from the kernel so that it can guarantee kernel code integrity even in the face of attacks against the kernel. Based on our TCB assumption, we can only rely on CPU-based protections to provide this isolation. CPU-based protections are based on the notion of privilege whereby more privileged software can modify both its own protections and those of less privileged software. Therefore, the enforcement agent must execute at a higher CPU privilege level than that of the kernel. We now describe how we design and build such an enforcement agent as a tiny hypervisor called SecVisor. SecVisor uses the virtualization features built into commodity CPUs to execute at the privilege level of a Virtual Machine Monitor (VMM).

The next issue that arises is: how to ensure the integrity of kernel code? SecVisor addresses this issue by ensuring that, when executing at the privilege level of the kernel (hereafter called the *kernel mode*), the CPU refuses to execute any code that is not approved by the user. In the interest of simplicity, henceforth, we will refer to SecVisor as approving the kernel code, with the understanding that SecVisor uses the user supplied policy for code approval. SecVisor does not prevent code from getting added to the kernel; only that the CPU will refuse to execute unapproved code. For example, the attacker could exploit a kernel-level buffer overflow vulnerability to inject code into the kernel's data segment. But the CPU will not execute the injected code since it is not approved by SecVisor. An additional requirement is that SecVisor approved code should not be modifiable by any entity on the system other than those in SecVisor's TCB and SecVisor. In order to implement these requirements, SecVisor needs to inform the CPU which code is approved for execution in kernel mode and also protect the approved code from modification. The CPU-based protections provide a natural way to address these. SecVisor sets the CPU-based protections over kernel memory to ensure that only code approved by it is executable in kernel mode and that the approved code can only be modified by SecVisor or its TCB.

All CPUs support at least one other privilege level (other than the kernel mode and VMM privilege level), called *user mode*, at which user programs execute. Given that a CPU will switch between user and kernel mode execution via control transfers, SecVisor needs to prevent the attacker from modifying the expected control flow of these control transfers to execute arbitrary code with kernel privilege. This requires two checks. First, SecVisor needs to ensure that the targets of all control transfers that switch the CPU to kernel mode lie within approved code. Without this, the attacker could execute arbitrary code with kernel privilege by modifying the targets of control transfers that enter kernel mode. Second, the control transfers that exit kernel mode to enter user mode must modify the privilege level of the CPU to that of user mode. Otherwise, the attacker could execute user programs with kernel privilege.

## 3.2 Required Properties for Approved Code Execution

We start designing SecVisor by casting the requirements into properties. Our first requirement is that the CPU only execute SecVisor approved code in kernel mode. Given that the CPU enters kernel mode from user mode, performs some processing in kernel mode, and exits kernel mode to return to user mode, SecVisor must provide the following three properties:

- **P1**: Every entry into kernel mode (where an entry into kernel mode occurs at the instant the privilege of the CPU changes to kernel mode) should set the CPU's Instruction Pointer (IP) to an instruction within approved kernel code.
- **P2**: After an entry into kernel mode places the IP within approved code, the IP should continue to point to approved kernel code until the CPU exits kernel mode.
- **P3**: Every exit from kernel mode (where we define an exit from kernel mode as a control transfer that sets the IP to an address in user memory) should set the privilege level of the CPU to user mode.

Our second requirement is that the approved code should only be modifiable by SecVisor and its TCB. Assuming that main memory can only be modified by code executing on the CPU or through Direct Memory Access (DMA) writes by peripheral devices, this requirement can be stated as:

- **P4**: Memory containing approved code should not be modifiable by any code executing on the CPU, but SecVisor, or by any peripheral device.

SecVisor uses hardware memory protections to achieve P2 and P4, as we describe next. Section 3.4 discusses how we achieve P1 by ensuring whenever the CPU enters to kernel mode it will start executing approved code, and P3 by intercepting and checking all kernel exits.

## 3.3 Using Hardware Memory Protections

The Memory Management Unit (MMU) and the IO Memory Management Unit (IOMMU) of the CPU enforce hardware memory protections. Then SecVisor must control all modifications to the MMU and IOMMU state. Since SecVisor executes at the privilege level of a VMM, it can virtualize the MMU and IOMMU. This enables SecVisor to intercept and check all modifications to the MMU and IOMMU state.

SecVisor uses page tables as the basis of its hardware memory protections. We choose page tables, rather than other MMU-based protection schemes, such as segmentation, because page tables are supported by a large number of CPU architectures. Using page tables requires SecVisor to protect the page tables from being modified by any entity but SecVisor and its TCB. There are two ways to achieve this. One, SecVisor can keep the page tables in its own address space and allow the kernel to read and modify them only via "safe" function calls. Two, SecVisor could virtualize physical memory. Virtualizing physical memory causes the addresses sent on the memory bus to be different from the physical addresses seen by the kernel. Hence, SecVisor must maintain page tables that translate the kernel's physical addresses to the physical addresses seen on the memory bus. These page tables could be kept in SecVisor's address space since the kernel is unaware of the virtualization of physical memory.

The choice of which of the above two methods to use illustrates the classic trade-off between performance on one hand and security and portability on the other. Using a function call interface is likely to be fast since there is no overhead of synchronizing the page tables maintained by SecVisor with the kernel's page tables (there is only one set of page tables). However, it increases the size of SecVisor's kernel interface which increases the security risk. It also requires modifications to the kernel's page table handling code which increases the amount of effort required to port a new kernel to SecVisor. On the other hand, virtualizing physical memory is likely to be slower due to the synchronization overhead. But it is certainly better for security and ease of portability of the kernel. Since our focus in this paper is on security and ease of portability we choose to virtualize physical memory. Henceforth, we will call the page table used by SecVisor to virtualize physical memory the *Protection Page Table.*

**Shared address space configuration.** In using the Protection Page Table to set protections over kernel memory, SecVisor has to consider how the kernel and user memories are mapped into address spaces. In most commodity OSes today, the kernel and user memories share the same address space. Such a shared address space configuration could enable an attacker to modify the control flow of the kernel to execute user code with kernel privilege. For example, the attacker could exploit a buffer overflow vulnerability and modify a return address stored on the kernel's stack to point to user code. To prevent these kinds of attacks, and thereby achieve property P2, SecVisor sets the Protection Page Table so that user memory is not executable when the CPU executes in kernel mode. On the other hand, it is clear that user memory has to be executable when the CPU is in user mode. Then, SecVisor has to intercept all transitions between kernel and user mode to modify the user memory execute permissions in the Protection Page Table. SecVisor uses the execute permissions themselves to intercept these transitions. It sets execute permissions in the Protection Page Table only for the memory of the mode that is currently executing. Then, all inter-mode transitions cause protection violations, which inform SecVisor of an attempted mode change via a CPU exception. Figure 1 illustrates how SecVisor manages memory permissions.

Given that SecVisor switches execute permissions on kernel entry and exit, a natural question arises: how are the execute permissions set initially? At system start-up, the kernel executes before user programs. Therefore, the SecVisor initializes the Protection Page Table so that only the approved kernel code is executable.

**W $\oplus$ X protections.** On each entry to kernel mode, SecVisor sets execute permissions in the Protection Page Table so that only approved code will be executable. Then, the CPU will generate an exception on every attempt to execute unapproved code in kernel mode. When SecVisor receives such an exception, it terminates the kernel. SecVisor also marks the approved code pages read-only in the Protection Page Table. This prevents any code executing on the CPU (except SecVisor) from modifying approved code pages, thereby satisfying part of property P4. From Figure 1 it can be seen that, in kernel mode, the pages of kernel memory will be ei-

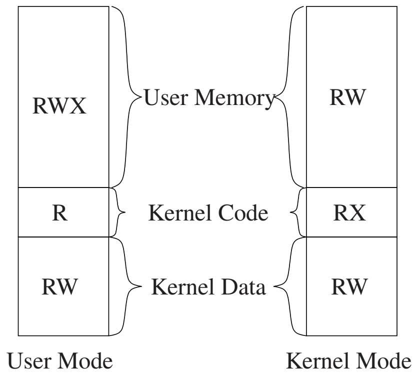


**Figure 1: Memory protections in the Protection Page Table for user and kernel modes. R, W, and X stand for read, write, and execute permissions, respectively. In case the kernel's permissions differ from those in the Protection Page Table, the actual permissions will be the more restrictive of the two. For example, it is likely that the kernel will mark its data segment to be read-only in user mode. Then, in user mode, the actual permissions over the kernel data segment would be R instead of the RW shown in the figure.**

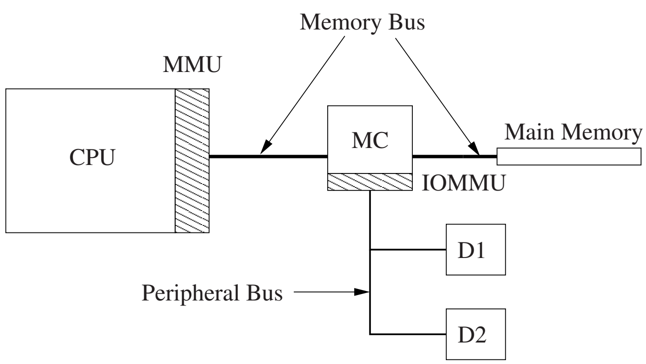


**Figure 2: System-level overview of memory protections used by SecVisor. MC is the memory controller, D1 and D2 are peripheral devices. The MMU enforces memory protections for accesses from the CPU while the IOMMU enforces DMA write protections.**

ther writable or executable, but never both. This type of memory protection is generally referred to as *$W \oplus X$ protection*.

**DMA write protections.** SecVisor uses the DMA write protection functionality of the IOMMU to protect approved code pages from being modified by DMA writes. These protections along with the read-only protections set in the Protection Page Table ensure that property P4 is satisfied. Figure 2 shows a system-level overview of the two hardware memory protections used by SecVisor.

## 3.4 Managing Kernel Entries and Exits

We now discuss how SecVisor ensures that kernel mode entries and exits satisfy properties P1 and P3.

**Kernel mode entries.** SecVisor ensures that all control transfers through which the CPU enters kernel mode will set the IP to an address within the approved code. This requires SecVisor to know the target of every possible control transfer through which the CPU can enter kernel mode. The key observation that allows us to find the target of every possible control transfer to kernel mode is that CPUs only allow kernel mode entries to transfer control to entry points designated by the kernel. This prevents user programs from triggering arbitrary control flows in kernel code by entering at arbitrary points. The kernel informs the CPU of the permitted entry points by writing the addresses of such entry points (hereafter called the *entry pointers*) in CPU registers and data structures like the interrupt vector table (IVT). Then, SecVisor only has to ensure that all entry pointers point to instructions within approved code to achieve property P1.

To find all the entry pointers, we need to identify all the CPU data structures that can contain entry pointers. By design, every CPU architecture has a set of control transfer events that trigger CPU execution privilege changes. Each control transfer event has an associated entry pointer in some CPU data structure. Therefore, our strategy to find all the entry pointers is to first create the exhaustive *entry list* of all control transfer events that can transfer control to kernel mode. The entry list can be created from the architectural specification of the CPU. Next, for each event in the entry list we find the CPU data structure which holds its entry pointer. In this manner, we obtain the list of all the CPU data structures which can hold the entry pointers.

SecVisor virtualizes the entry pointers and only permits the kernel to operate on the virtualized copies. This allows SecVisor to intercept and check all modifications to the entry pointers. The virtualization can be performed in two ways. First, SecVisor can provide the kernel with "safe" function calls through which the kernel can read and modify the entry pointers. Second, SecVisor can maintain *shadow* copies of the entry pointers for use by the CPU, and keep the shadow copies synchronized with the kernel's entry pointers. As with virtualizing physical memory, the choice between these two alternatives is a trade-off of performance versus security and portability. We prefer the shadowing method because it reduces the size of SecVisor's kernel interface and also reduces the number of changes required to port a kernel to SecVisor.

**Kernel mode exits.** All legitimate methods that exit kernel mode will transfer control to code in user memory. If on each entry to kernel mode the CPU will start executing approved code, i.e., property P1 is satisfied, it is fairly direct to ensure that exits from kernel mode will set the CPU privilege to user mode (property P3).

Recall from Figure 1 that SecVisor marks kernel memory non-executable in user mode. If property P1 is satisfied, all kernel mode entries will try to execute approved code, which is part of kernel memory. This will cause the CPU to generate an exception. As part of handling this exception, SecVisor marks all user memory non-executable. Thus, any exit to user mode will cause a protection violation, generating a CPU exception. As part of handling this exception, SecVisor sets the privilege level of the CPU to user mode.

# 4. BACKGROUND

Before we present the details of SecVisor's implementation, we present some background on the features of x86 necessary to understand the implementation. We describe the x86 memory protection mechanisms followed by the different control transfer events that can be used to enter and exit kernel mode. Finally, we describe the Secure Virtual Machine (SVM) extensions, which are present in recent AMD x86 CPUs. Readers who are familiar with this material may wish to skip directly to Section 5.

## 4.1 Overview of x86 Memory Protections

This section gives a brief overview of the two memory protection mechanisms of the x86 CPU: segmentation and paging.

In 32-bit mode, segment-based protections are based on four privilege levels called rings. Ring 0 is the most privileged while Ring 3 is the least privileged. The current execution privilege level of the CPU is stored in the (architecturally-invisible) CPL register. Segment-based protections divide up the memory into variable size regions called segments. Each segment of memory has a descriptor associated with it. This descriptor contains various attributes of the segment such as the segment base address, the segment size, and the segment access permissions. The descriptors are stored in two tables called the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT). The CPU has two registers called the `gdtr` and `ldtr` that contain the addresses of the GDT and LDT, respectively.[1] Software can refer to descriptors in the GDT and LDT by storing their indices in the CPU's *segment registers*. There are six segment registers: `cs, ds, es, fs, gs,` and `ss`. Of these the `cs` segment register holds the index of the descriptor of the code segment that the CPU is currently executing from. The `ds, es, fs,` and `gs` segment registers hold indices of descriptors of data segments while the `ss` segment register holds the index of the stack segment's descriptor.

Page-based protections divide the virtual address space of the CPU into pages of fixed size. Page table entries describe the access permissions of each page. Per-page execute permissions are supported by the CPU only when the Physical Address Extensions (PAE) paging mode is used. The CPU has a set of registers called the Control Registers which allow software to control various aspects of the MMU. In particular, the control register `cr0` has two bits called `pe` and `pg` that allow software to turn memory protections on/off, and turn paging on/off, respectively. `cr0` also has the `wp` bit that controls whether the CPU enforces read/write permissions in both user and kernel mode or only in user mode. When `wp` bit is cleared, read/write permissions are enforced only in user mode. Setting this bit, causes the CPU to enforce read/write permissions in both user and kernel modes. The control register `cr3` holds the physical address of the page tables, while `cr4` has the `pae` bit which turns the PAE mode on/off.

## 4.2 Control Transfer Events on the x86

We now give an overview of the control transfer events on the x86 platform that perform inter-ring switches. We also mention the CPU registers and data structures that hold the corresponding entry pointers.

An x86 CPU assumes that control transfers between rings always originate at a ring of lower privilege. In other words, a lower privilege ring *calls* a higher privilege ring, which *returns* to the lower privilege ring. Then, on the x86 architecture, the *exit list* of events that can cause the CPU to exit kernel mode contains the *return* family of instructions: `ret, iret, sysexit,` and `sysret` [2].

The entry list consists of the hardware interrupts and the exceptions, and the instructions in the *call* family: `jmp, call, sysenter, int` (software interrupt), and `syscall` [2]. The entry pointers for hardware interrupts and exceptions, and software interrupts are located in the interrupt descriptor table (IDT). The CPU has a register called the `idtr` which holds the address of the IDT. We now briefly describe the remaining instructions in the entry list: the `jmp, call, sysenter,` and `syscall`.

**jmp and call.** The entry pointers for the `jmp` and the `call` instructions exist in the GDT and the LDT. The x86 architecture does not allow the `call` and `jmp` instructions to directly specify a higher

[1]This is a slight simplification. The `ldtr` actually stores the selector of the LDT descriptor in the GDT. The LDT descriptor in the GDT contains the LDT's base address.

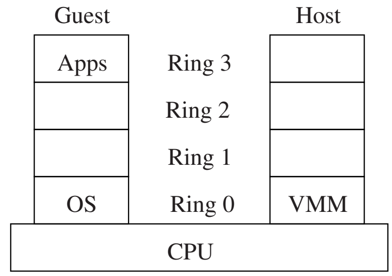


**Figure 3: A VMM and one guest VM executing on a CPU with SVM extensions. Both the VMM and the guest VM have access to all four x86 CPU privilege levels. Within the guest VM, the OS executes at the highest CPU privilege level, while user applications execute at the lowest CPU privilege level.**

privilege code segment as the target. Instead, these instructions must use one of three data structures, the *call gate*, the *task gate*, or the *task descriptor*, as their target. The kernel is expected to set up these data structures with the addresses of acceptable entry points. The CPU then ensures that the `jmp` and the `call` instructions will transfer control only to entry points permitted by the kernel. The task gates and the call gates can reside in the GDT or the LDT. Task descriptors can only reside in the GDT.

**sysenter and syscall.** `Sysenter` and `syscall` are special instructions that decrease the latency of system calls. The entry pointers for the `sysenter` and the `syscall` instructions are in CPU registers called the model specific registers (MSR).

The `sysenter` instruction uses the MSRs `msr_sysenter_cs` and `msr_sysenter_eip` for its entry pointer. When a user mode program executes the `sysenter` instruction, the CPU loads the `cs` segment register and the IP from the `msr_sysenter_cs` and `msr_sysenter_eip` respectively. The `syscall` instruction was recently introduced as replacement for `sysenter`. The use of this instruction is enabled by setting the `efer.sce` bit of the MSR `efer`. It uses the `msr_star` for its entry pointer. On the execution of the `syscall` instruction, the CPU loads `cs` and the IP with bits 47-32 and bits 31-0 of the `star` respectively.

## 4.3 Overview of AMD's SVM extensions

AMD provides CPU-based virtualization support via the SVM extensions. SVM separates the CPU execution into two modes called guest mode and host mode. The VMM (also referred to as host) executes in host mode while all the virtual machines (VM) (also referred to as guests) execute in guest mode. The host and guest modes have separate address spaces. Software executing in both modes can execute in any of the four privilege levels. For example, under SVM, it is possible for both the guest OS and the VMM to execute at the highest CPU privilege level. Figure 3 shows one possible execution configuration of the host and guest modes when using SVM extensions.

Each guest has a data structure called the Virtual Machine Control Block (VMCB) associated with it, which contains the execution state of the guest. The value of the CPL register is a part of this execution state. To execute a guest, the VMM calls the `vmrun` instruction with the VMCB as the argument. The CPU then loads the execution state of the guest from the VMCB and begins executing the guest. Once started, the CPU continues to execute the guest until an event that has been intercepted by the VMM occurs. On hitting an intercept, the CPU suspends the execution of the guest, stores the guest's execution state in the VMCB, and exits to the

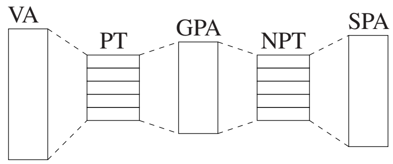


**Figure 4: Operation of the nested page tables. VA is the virtual address space, PT are the guest kernel's page tables, GPA is the guest's physical address space, NPT are the nested page tables, and SPA is the system physical address space.**

host. The host then executes until it resumes a guest using `vmrun`.

**Intercepts.** Intercepts allow the host to intercept various CPU events that occur during the execution of the guest. The host indicates which events in the guest it wants to intercept by setting bits in the VMCB. Through intercepts, the host executes at a higher CPU privilege than the guests because it can control what operations the guests are allowed to perform. SVM defines four kinds of intercepts: instruction intercepts, interrupt and exception intercepts, IO intercepts, and MSR intercepts. Instruction intercepts allow the host to intercept the execution of various instructions. Interrupts and exceptions delivered to the guest are intercepted by means of the interrupt and exception intercepts. SVM has a mechanism called *event injection* that allows the host to pass the intercepted interrupts and exceptions to the guest. IO intercepts are used by the host to intercept reads and writes of x86 IO ports by the guest. Finally, the MSR intercepts allow the host to intercept guest reads and writes to MSRs.

**Device Exclusion Vectors (DEV).** SVM provides support for a limited IOMMU by modifying the memory controller to add DMA read and write protection support for physical memory on a per page basis. This protection is enabled through the use of Device Exclusion Vectors (DEV), which are bit vectors with one bit for each 4 Kbyte physical page. If the bit corresponding to a physical page is set, the memory controller disallows any DMA reads from or DMA writes to that page.

The DEV mechanism is controlled through a set of configuration registers which are mapped to the Peripheral Component Interconnect (PCI) Configuration Space. Software can read and write registers in the PCI Configuration Space using two I/O ports called the Configuration Address Port and the Configuration Data Port. Software writes the address of the register it wants to access to the Configuration Address Port and reads or writes the register by reading or writing the Configuration Data Port.

**Nested page tables (NPT).** Since the host virtualizes physical memory, the physical addresses used by a guest could be different from the physical addresses that are sent on the memory bus. Therefore, the host needs to translate the guest's physical addresses to the physical addresses sent on the memory bus (hereafter called the *system physical addresses*). SVM provides nested page tables (NPT) for this purpose. That is, the NPT provide CPU-supported physical memory virtualization. The NPT is maintained by the host; the guest kernel maintains its own page tables to translate virtual addresses to guest physical addresses. This two step translation from virtual to system physical addresses is illustrated in Figure 4. Note that the NPT is used only when the CPU is executing in guest mode. When executing in host mode, the CPU will use the page tables of the host since physical memory is not virtualized in host mode.

Accesses to physical memory pages are subjected to permission checks in both the NPT and guest kernel page tables. In particular, a page is writable only if it is marked writable both in the kernel page table and in the NPT. Similarly, the contents of a page are executable only if the page is marked executable in both the kernel page table and in the NPT. The CPU generates a Nested Page Fault exception and exits to the host on any NPT protection violation.

The NPT mechanism also provides separate copies of the all control registers for the host and the guest. The `nested_cr3` register contains the pointer to the NPT. The guest control registers control the MMU configuration for address translation between virtual and guest physical addresses and the host control registers control guest physical to system physical address translation.

**ASID support.** SVM adds Address Space Identifier (ASID) bits to the Translation Lookaside Buffer (TLB) entries in order to allow the CPU to distinguish between the TLB entries of the different address spaces (the host and guests) that can co-exist due to virtualization. Tagging the TLB entries with the ASID eliminates the need to flush the TLB when switching address spaces.

**Late launch.** Late launch is a capability of SVM that allows the CPU to execute an arbitrary piece of code in isolation from all entities on the system, but the CPU, the memory controller, and the memory chips. A late launch can be invoked at any time during the operation of the system. If the system has a Trusted Platform Module (TPM) chip, late launch also allows an external verifier to verify that the correct code was invoked for execution.

# 5. IMPLEMENTATION USING AMD SVM

In this section, we discuss how we realize the SecVisor design described in Section 3 on a system that has an AMD CPU with SVM extensions. We first describe how SecVisor protects its own memory. Then we discuss physical memory virtualization in SecVisor, followed by the protection of the DEV mechanism virtualization. Finally, we describe how SecVisor handles kernel entry and exit. An important thing to remember while reading this section is that SecVisor is a hypervisor (and not a VMM) and supports only one guest (hence the reference to *the kernel* in what follows).

## 5.1 Allocating and Protecting SecVisor Memory

SecVisor executes in SVM host mode while the kernel and applications execute in guest mode. Thereby SecVisor executes at a higher CPU privilege level than the kernel. It intercepts events in the kernel and applications to virtualize the MMU, the IOMMU, and physical memory.

Using the host mode gives SecVisor its own address space, which simplifies protection of SecVisor's memory. SecVisor ensures that its physical memory pages are never mapped into the Protection Page Table. Since the Protection Page Table is maintained by SecVisor it is simple to check that the above condition holds. Also, SecVisor uses the DEV mechanism to protect its physical pages against DMA writes by devices.

The question of which physical pages SecVisor should allocate for its own use requires consideration. The main issue here is that of handling DMA correctly. In a system with SecVisor, all DMA transfers set up by the kernel use guest physical addresses to specify the source and destination. Since SecVisor virtualizes physical memory, the guest physical addresses can be different from the system physical addresses. Therefore, guest physical addresses of the DMA transfers need to be translated to system physical addresses for DMA transfers to work correctly. The ideal solution to this problem is to use an IOMMU that will translate the guest physical addresses used by a device during DMA to system physical addresses. SecVisor only needs to ensure that the IOMMU uses the correct Protection Page Table. However, SVM currently does not provide such an IOMMU facility. In the absence of hardware support, SecVisor could intercept all DMA transfer setup performed by the kernel in order to translate between guest and system physical addresses. However, intercepting DMA transfer setup is not simple. It depends heavily on the design of the kernel as it requires the kernel to call SecVisor as part of each DMA transfer setup. Hence we prefer not to use this method, given our desire to reduce the size of SecVisor's kernel interface and minimize the changes required to port the kernel.

Instead, SecVisor circumvents the whole issue of translating addresses for DMA by making sure that the guest to system physical address mapping is an identity map. To achieve the identity mapping, SecVisor allocates its physical memory starting from the top of the installed RAM.[2] The kernel uses all memory from address zero to the start of SecVisor's physical memory. SecVisor informs the kernel of the reduction in physical memory available by passing a command line parameter at kernel boot.

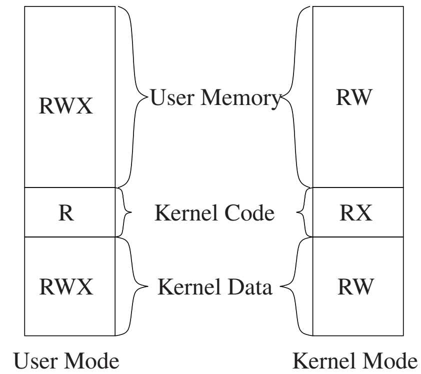


**Figure 5: NPT based memory protections for user and kernel modes. R, W, and X stand for read, write, and execute permissions, respectively.**

## 5.2 Virtualizing the MMU and Memory

We now discuss how SecVisor virtualizes the physical memory to set page-table-based memory protections. The details depend on whether we use a software or a hardware method to virtualize physical memory. The software virtualization uses shadow page tables (SPT) as the Protection Page Table, and the hardware virtualization uses the SVM NPT. Even though the NPT offers better performance, we also implement SPT support in SecVisor because current x86 CPUs from AMD do not have support NPT. According to AMD, suitable CPUs should be available in Fall 2007.

**Hardware memory virtualization.** SVM's nested paging facility provides a second set of page tables (the NPT) that translate guest physical addresses to system physical addresses (Figure 4). The NPT is very well suited for setting page-table-based protections both from a performance and security perspective due to two reasons.

First of all, the design of SVM ensures that access permissions of a physical page are the more restrictive of those in the kernel's page tables and the NPT. Therefore, SecVisor uses the NPT to set its memory protections, without any reference to the kernel's page tables. It is easy for SecVisor to isolate the NPT from accesses by the guest since the existence of the NPT is transparent to the guest. SecVisor isolates the NPT by allocating physical pages from its own memory for the NPT. Since SecVisor's physical pages are never accessible to the guest and they are protected against DMA writes, the NPT is inaccessible to everything but SecVisor and SecVisor's TCB.

Secondly, the nested paging facility eliminates the need for SecVisor to intercept kernel writes to the MMU state. It provides the guest and host with their own copies of the Control Registers, which control MMU state. Since SecVisor only uses the NPT to set its protections, it can allow the kernel (guest) to freely modify the guest control registers. Put another way, with nested paging, SecVisor can virtualize the MMU without intercepting kernel writes to the control registers. Also, since the contents of the NPT are completely independent from those of the kernel's page tables there is no need for SecVisor to update the NPT when the kernel makes changes to the kernel's page tables. Clearly, both of these factors result in better performance and decrease the code size of SecVisor.

The only drawback of using the NPT is that the kernel needs to pass guest physical addresses rather than virtual addresses in its requests to SecVisor to change memory permissions. However, this requirement for address translation is unlikely to be a performance bottleneck since this is not a frequent event (modification of memory permissions only needs to be done when kernel modules are loaded or unloaded). Also, passing guest physical addresses does not require any modifications to the Linux kernel since it already has functions to translate between virtual and physical addresses.

As mentioned in Section 3.3, there are two tasks that SecVisor accomplishes via page-table-based protections. One, it sets W $\oplus$ X protections over kernel memory when executing in kernel mode. Two, it modifies the execute permissions of user and kernel memory depending on whether the CPU executes in kernel or user mode. Both tasks are easily accomplished using the NPT.

To set the W $\oplus$ X protections, SecVisor maintains a list of guest physical pages that contain approved code. The kernel can request modifications to this list. Any requests to add new entries in the list must be approved by the user-supplied approval policy. When executing in kernel mode, SecVisor clears the no-execute (NX) permission bit only for the NPT entries of guest physical pages in the list (Figure 5).

Modifying execute permissions over user and kernel memory requires SecVisor to know which guest physical pages contain the kernel's data segment and which are user pages. SecVisor could maintain a list of guest physical pages that belong to the kernel's data segment similar to that for the kernel code. However, adopting this design is likely to degrade performance since the pages frequently move between the kernel data segment and user space. Therefore, we adopt a different design.

When the CPU executes in user mode, SecVisor marks all guest physical pages except those containing approved code executable in the NPT. Note that this does not open an avenue for attacks that could execute kernel data segments in kernel mode since property P1 guarantees that all control transfers to kernel mode will set the IP to an address within approved code, and SecVisor satisfies property P1 using a different mechanism than the NPT (by ensuring that the entry pointers all point to approved code). Note that SecVisor still makes the approved kernel code non-executable during user mode execution so that all transitions from user mode to kernel mode can be easily intercepted via nested page faults.

To keep the size of the NPT handling code small and to reduce the latency of switches between user and kernel mode, SecVisor maintains two NPTs, one for address translations during user mode execution and the other for address translations during kernel mode

[2]The addresses at the very top of RAM are used by the ACPI code. SecVisor allocates its space starting from just below the ACPI region.

execution. These two NPTs set different permissions on user and kernel memory as Figure 5 shows. The synchronization costs of maintaining two NPTs are not high since the NPTs need to be modified only when kernel code is changed.

On each transition from user mode to kernel mode or vice versa, SecVisor changes the `nested_cr3` field in the VMCB to point to the NPT of the mode that is going to execute next. In order to avoid flushing the TLB as part of these transitions, SecVisor associates the two NPTs with different ASIDs. The drawback of doing this is increased data TLB pressure due to the fact that the translation for the same virtual address could exist in the data TLB under two different ASIDs. The instruction TLB does not suffer from this problem since user mode will never execute kernel code and vice versa. We use different ASIDs for the two NPTs under the assumption that the performance benefits of not having to flush the TLB on every transition between user and kernel modes should be greater than the performance degradation due to the increased data TLB pressure.

**Software memory virtualization.** We now describe SecVisor's software memory virtualization technique based on shadow page tables (SPT). A SPT virtualizes memory by maintaining the mapping between virtual and system physical addresses. Therefore, the SPT needs to be kept synchronized with the kernel's page tables, which translate virtual addresses to guest physical addresses. Using an SPT-based approach incurs both a code size increase and performance penalty compared to a NPT-based implementation, mainly due to this synchronization requirement.

The SPT implementation in SecVisor uses a single SPT for both user and kernel mode execution. As with the NPT, SecVisor protects the SPT by allocating physical pages for it from SecVisor's memory. SecVisor also keeps the SPT synchronized with the current kernel page table. Having a single SPT increases the cost of transitions between the user and kernel modes since execute permissions over user and kernel mode have to be modified on each transition. In spite of this, we do not use an SPT each for user and kernel mode due to fact that SPTs need to modified far more frequently than NPTs. Unlike the NPTs which only need to modified on changes to kernel code, the SPT needs to modified whenever the kernel makes modifications to its current page table (for example, on a page fault) or when it makes another page table current (as part of a context switch). Having to synchronize two SPTs with the kernel's page table would double the number of memory writes needed for the frequently used SPT synchronization operation.

Like the NPT, SecVisor performs two operations on the SPT: sets W $\oplus$ X protections over kernel memory and modifies execute permissions over user and kernel memory on each mode transition. Figure 6 shows how SecVisor sets protections in the SPT for user and kernel mode execution. One point to note is that when SecVisor uses shadow paging, the SPT are the only page tables used by the CPU. Therefore, (unlike the NPT) the permissions that SecVisor sets in the SPT must be the more restrictive of its own permissions and those of the kernel.

SecVisor needs to modify execute permissions of user and kernel memory so that all mode transitions cause page faults. To minimize the overhead of modifying the execute permissions on each transition between user and kernel modes, SecVisor uses the NX bits in the page table entries in the second level of the page table hierarchy (the first level entries do not have NX bits). This optimization allows SecVisor to switch execute permissions by changing the NX bits in only the 4 second level page tables. Since each page table is 4 Kbytes in size the maximum amount of data accessed by this operation will be 16 Kbytes.[3]

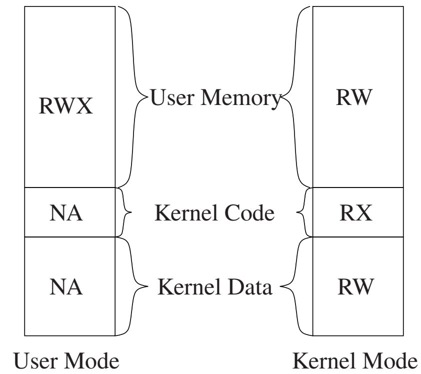


**Figure 6: SPT based memory protections for user and kernel modes. R, W, and X stand for read, write, and execute permissions, respectively. NA stands for Not Accessible. The permissions shown correspond to the more restrictive of those set by the kernel and by SecVisor.**

Figure 7 shows the different guest operations that SecVisor needs to intercept in order to synchronize the SPT with the kernel's page tables. The `cr3` register holds the pointer to the page tables. Therefore, the kernel will write to `cr3` when it wants to use a new page table. SecVisor intercepts this write and copies the new kernel page table into the SPT. The `invlpg` instruction is used to invalidate a single TLB entry. When the kernel modifies an existing page table entry it must invalidate the corresponding TLB entry. Intercepting the execution of `invlpg` enables SecVisor to synchronize the SPT with modified kernel page table entry. Finally, when the kernel creates a new entry in its page tables, and attempts to use it, it will cause a page fault since the corresponding entry will not exist in the SPT. SecVisor uses such *shadow page faults* to synchronize the SPT by copying the newly created kernel page table entry.

The current SPT synchronization code of SecVisor uses a very simple design that trades off performance for security and ease of porting a kernel. For example, the synchronization code does not try to aggressively batch the synchronization of the SPT in order to amortize synchronization costs. On the other hand, we do not need to make any modifications to the kernel's page table handling code.

During SPT synchronization, SecVisor copies the kernel page table entries into the SPT. The kernel page table entries can be modified by the attacker. The attacker's modifications must not propagate to the SPT fields that enforce approved kernel mode code execution. To ensure this, SecVisor never copies the read/write, and execute permission bits from the kernel page table during SPT synchronization. Additionally, it performs to checks on the address fields of the kernel page table entries.

First, SecVisor checks for and prevents virtual address aliasing of approved code physical pages. If this check is not performed, the attacker could modify the contents of approved code pages using the following attack. The attacker could create, in the kernel page table, a writable virtual alias to a physical page containing approved code. SecVisor's SPT synchronization code, without an aliasing check, will copy the attacker's page table entry into the SPT. The attacker could then modify the contents of the approved code physical page by writing to it using virtual addresses corresponding the writable virtual alias it created.

[3]Modifying NX permissions of the approved code pages alone does not seem to benefit performance due to the extra state that has to be maintained to do this.

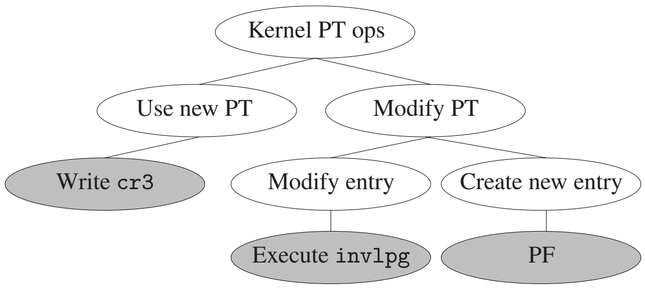


**Figure 7: Design of SecVisor SPT synchronization code. Each non-leaf node indicates an operation that the kernel can perform on its page tables. The leaf nodes (shaded) indicate the guest event that SecVisor must intercept to learn of the corresponding kernel operation. PT and PF stand for Page Table and Page Fault respectively.**

Second, SecVisor needs to check if the kernel page tables map approved code virtual pages to physical pages containing approved code. Without this check, the attacker could execute arbitrary code in kernel mode. The attacker could do this by modifying the kernel page table entries that translate the virtual addresses of approved code to point to physical pages that contain the attacker's code. If SecVisor copied the kernel's page table into the SPT without checking the physical address contained in the entries that translate virtual addresses of approved code, the CPU would would then execute the attacker's code instead of the approved code.

To perform the above two checks, SecVisor internally maintains the list of virtual page to physical page translations for approved code pages. When synchronizing the SPT with the kernel's page tables, SecVisor consults this list to detect if any entry in the kernel's page table is an alias of an approved code physical page. If virtual address aliasing is detected, SecVisor does not copy aliasing kernel page entry. Instead it marks the corresponding SPT entry invalid. Also SecVisor does not copy the page table entries that translate virtual addresses of approved code into the SPT from the kernel page table. Instead, it sets these entries based on the entries in its internal list of approved code pages.

The final issue that needs to be dealt with in regard to SecVisor's SPT implementation is how SecVisor controls kernel modifications to the MMU state. In order to do this, SecVisor intercepts writes to the `cr0` and `cr4` registers. These intercepts enable SecVisor to prevent kernel writes to the `pe`, `pg`, `wp`, and `pae` bits.

## 5.3 Protecting the DEV Mechanism

As we mentioned in Section 4, SVM provides the DEV mechanism to control DMA access to physical memory pages. Guest software and devices need to be prevented from modifying both the DEV bit vector and the DEV configuration registers in the PCI configuration space. SecVisor protects the DEV bit vector in the same manner it protects the SPT and the NPT: by allocating physical pages for the bit vector from its own memory. By design, the memory controller blocks all accesses from devices to the DEV PCI configuration space. SecVisor protects the DEV configuration registers against writes by guest software by virtualization.

The I/O intercept mechanism of SVM provides a convenient way for SecVisor to virtualize the DEV configuration registers. SecVisor intercepts all writes to the Configuration Data Port. The I/O intercept handler in SecVisor figures the target of the write by looking at the the address in the Configuration Address Port and blocks any write by guest software to the DEV configuration registers. If the write is going to any other PCI configuration register the I/O intercept handler performs the write on behalf of the guest software.

## 5.4 Kernel Mode Entry and Exit

We now describe how SecVisor achieves properties P1 and P3 on the x86 architecture. Property P3 requires that all kernel mode exits set the privilege level of the CPU to that of user mode. In the case of Linux executing on a x86 CPU, user programs execute in Ring 3. Then, on kernel exit SecVisor must set the privilege level of the CPU to Ring 3. As we already pointed out in Section 3.4, as far as all kernel entries set IP to approved code (property P1), all kernel mode exits will cause a protection exception. As part of handling this exception SecVisor sets the CPL field of the VMCB to 3, thereby ensuring that when the guest resumes execution, the CPU will execute in Ring 3.

SecVisor ensures that all CPU entries into kernel mode will satisfy property P1 (IP will point to approved code at entry) by checking that all entry pointers point to approved code. From Section 4.2, we see that the entry pointers all exist in the GDT, the LDT, the IDT, and some MSRs (for `syscall` and `sysenter`). Then, from Section 3.4, we need to maintain shadow copies of the three tables and the relevant MSRs to satisfy property P1. By maintaining these shadow copies SecVisor ensures that the entry pointers used by the CPU all point to approved code. In the rest of this section, we describe how SecVisor maintains the shadow copies of the different entry pointers on the x86.

Maintaining shadow copies of the MSRs is simple since SVM provides facilities for intercepting read and write to each MSR. SecVisor sets bits in the VMCB to intercept writes to the MSRs `msr_sysenter_cs`, `msr_sysenter_ip`, and the `msr_star`. The intercepts enable SecVisor to check whether the entry pointers the kernel writes to these MSRs point to approved code.

Shadowing the GDT, LDT, and IDT is somewhat more involved since our goal is to check and write-protect not only the CPU pointers to the GDT, LDT, and IDT (the `gdtr`, `ldtr`, and `idtr`) but also the contents of the tables themselves. While SVM provides facilities to intercept writes to the `gdtr`, `ldtr`, and `idtr`, the tables themselves exist in memory and need to be write-protected via shadowing.

However, merely maintaining shadow copies of all entry pointers is insufficient to guarantee property P1. To see why this is so, first note that the entry pointers all contain virtual addresses. Then note that the approved code itself is resident in physical pages. Therefore the page tables must translate the virtual addresses of the entry pointers to physical addresses corresponding to approved code. Otherwise the attacker could violate property P1 by having the virtual addresses of the entry pointers point to physical pages of its choice.

In view of the above, SecVisor needs to maintain shadow copies of the GDT, LDT, and IDT to write-protect the virtual addresses of the entry pointers. It also needs to check and write-protect the page table entries that translate the virtual addresses of the entry pointers to physical addresses. Also, note that the page table entries that translate the virtual addresses present in the `gdtr`, `ldtr`, and `idtr` also need to be similarly checked and write-protected.

In what follows, we first discuss how SecVisor synchronizes the shadow copies of these tables with their kernel counterparts. Then, we tackle the issue of checking and write-protecting the shadow tables as well as the page table entries that translate the virtual addresses of the entry pointers, and the virtual addresses present in the `gdtr`, `ldtr`, and `idtr`. In the discussion that follows, we call these page table entries the *critical page table entries*.

To deal with the synchronization issue, we observe that the shadow copies of these tables only need to control execution in user mode since property P1 deals with transition from user mode to kernel mode. In other words, during kernel mode execution the CPU can use the kernel's GDT, LDT, and IDT. This observation enables two simplifications. One, we can implement a *lazy synchronization* scheme to maintain shadow copies of the GDT, LDT, and IDT. This lazy synchronization scheme only synchronizes the shadow tables when the CPU transitions from kernel to user mode. Since all legitimate modifications to these tables can only occur in kernel mode, the lazy synchronization allows SecVisor to batch all its updates to the shadow copies. As part of the synchronization, SecVisor checks that all entry pointers in the shadow GDT, LDT, and IDT contain virtual addresses of approved code. Two, SecVisor does not need to intercept writes to the `gdtr, ldtr,` and `idtr`. Since the shadow tables needs to be in control of execution only in user mode, SecVisor can set these registers to point to the shadow copies as part of the kernel mode to user mode transition. This it does by changing the corresponding values in the VMCB as part of handling a kernel to user mode transition. Any attempt by the user programs to modify any of these registers will result in the CPU throwing an exception. Since the corresponding entry pointer is pointing to approved code, this exception will cause approved kernel code to be invoked, satisfying property P1.

From the description of our lazy synchronization scheme and from the requirement that kernel exits throw exceptions, it can be seen that a circular dependency exists. The lazy synchronization scheme requires each exit from kernel mode to cause an exception so that the shadow GDT, LDT, and IDT can be synchronized with the corresponding tables in the kernel. On the other hand, ensuring that an exception will occur on kernel mode exit requires that the shadow GDT, LDT, and IDT contain valid entry pointers since user mode will be marked non-executable only if the control returns to approved code on kernel entry. We break this circular dependency by setting up an initial condition. We note that at system start-up the kernel executes before user programs. SecVisor sets protections at system start-up that make all user memory non-executable. Thereby, the first exit from kernel mode will axiomatically cause a CPU exception. As part of handling this exception SecVisor synchronizes the initial shadow GDT, LDT, and IDT with the corresponding tables in the kernel.

We now take up the issue of write-protecting the shadow tables and the critical page table entries, starting with the shadow tables. Note that, due to our lazy synchronization scheme, the shadow tables only need to be write-protected when the CPU executes in user mode. The simplest way to do this seems to be to not map the shadow tables into the guest's address space at all (like we do for the SPT and the NPT). However, not mapping the shadow tables into the guest's address space increases the code size and complexity of SecVisor and also decreases performance. The preceding claim can be justified by the observation that if the shadow tables are not mapped into the guest's virtual address space then SecVisor will have to simulate every one of the many CPU events in user space that access these tables. For example, hardware interrupts require the CPU to read the address of the interrupt handler from the IDT. If the IDT is not mapped into the guest's virtual address space, the CPU will generate a page fault. SecVisor could intercept this page fault to learn of the occurrence of the interrupt. Then, it would have simulate the delivery of the interrupt to the guest increasing the complexity and code size of SecVisor.

To simplify the design of SecVisor, we keep the shadow tables mapped in the guest's virtual address space. SecVisor needs a contiguous range of virtual addresses in the guest's virtual address space in order to perform the this mapping. When the kernel boots, SecVisor requests it to permanently allocate 256 Kbytes of contiguous virtual addresses within kernel memory for holding the shadow tables and to statically map this range of addresses of a contiguous range of guest physical addresses. In the discussion that follows, we call this region the *shadow table area*. SecVisor maintains the shadow GDT, LDT, and IDT in shadow table area and write-protects this area to prevent writes when the CPU executes in user mode. The exact method SecVisor uses to write-protect the shadow table area depends upon whether it uses shadow paging or nested paging to virtualize physical memory.

The critical page entries that need to be write-protected depend on whether shadow or nested paging is used to virtualize physical memory. With shadow paging, the SPT are the only page tables used by the CPU. Therefore, SecVisor only needs to check the critical page table entries in the SPT since the SPT is inaccessible to the guest. With nested paging, the CPU uses both the kernel page tables and the NPT for address translation, as shown in Figure 8. In this case then, the critical page tables entries exist both in the kernel page tables and in the NPT. In what follows, we first describe how SecVisor write-protects the shadow area and the critical page table entries with shadow paging. Then, we describe how SecVisor handles the case of nested paging.

**Using shadow Page Tables.** With shadow paging, write-protecting the shadow area is straightforward: SecVisor sets up read-only mappings in the SPT entries that map the virtual address of the shadow table area to system physical addresses. Recall from Section 5.1 that the guest to system physical address mapping is the identity map. Therefore, SecVisor fills the address field of the SPT entries mapping the shadow table area with the guest physical addresses of the shadow table area. Since the virtual and guest physical addresses of the shadow table area do not change during the system lifetime, the SPT entries for the shadow table area need to be set only once.

Protecting the critical page table entries is also simple since the SPT is inaccessible to the guest. SecVisor only needs to check if the critical page table entries are set up correctly in the SPT. In other words, SecVisor needs to check that all entry pointers translate to physical pages containing approved code and that virtual addresses of the `gdtr, ldtr,` and `idtr` to be used in user mode translate to physical addresses of the shadow tables in the shadow area. SecVisor performs this check as part of its lazy synchronization scheme that synchronizes the shadow tables with the kernel's copies.

**Using nested Page Tables.** When using nested paging, SecVisor can write-protect the shadow table area by marking the corresponding guest physical pages read-only in the NPT and by checking and protecting all the critical page table entries. In this case, the critical page table entries exist both in the kernel page tables and in the NPT. Since the NPT exists in SecVisor's memory and guest to system physical address mapping is an identity map, validating and write-protecting the critical page table entries in the NPT is trivial.

To see how SecVisor protects the critical page table entries in the kernel's page table, observe that the CPU accesses the kernel's page tables using guest physical addresses. Then, the kernel's page table could be protected from writes by guest user mode software by simply removing the write permissions in the NPT entries for the guest physical addresses of the kernel's page tables. Also, the DEV bit vector needs to protect the physical pages that contain the kernel's page table from DMA writes. Adopting this approach potentially requires that the NPT used for user mode execution and the DEV bit vector be modified on each context switch since the page tables of each user process could use different physical pages. Modifying the NPT will require the TLB to be flushed. Modifying

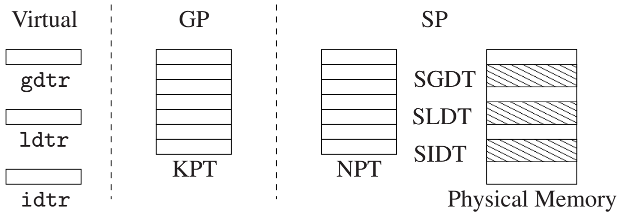


**Figure 8: Sequence of address translations required to access the shadow GDT, LDT, and IDT (SGDT, SLDT, and SIDT in the figure) using the virtual addresses stored in the `gdtr`, `ldtr`, and `idtr`. KPT and NPT are the kernel and nested page tables respectively. SP refers to system physical address space and GP to guest physical address space.**

the DEV bit vector will require the invalidation of DEV cache in the memory controller. This invalidation requires software to set a bit in the DEV configuration registers and monitor the bit until it is cleared by the hardware. The clearing of the bit indicates that the hardware has invalidated the DEV cache. For performance, we would like to avoid performing several memory writes to the NPT, a TLB flush, and a DEV bit vector modification in SecVisor on each context switch.

Alternately, we could copy the critical entries in the kernel's page table into the shadow table area. Since the shadow table area exists at the same guest physical address, and hence system physical address, for the entire lifetime of the system both the NPT and DEV bit vector protections need to be set only once. However, this solution might also requires several memory writes due to the multi-level page table hierarchy of the x86. In view of the multi-level page table hierarchy, SecVisor needs to protect the page tables at all levels of the hierarchy that translate the virtual addresses of the shadow table area or the entry pointers to physical addresses. This could require several page tables to be copied into the shadow table area to write-protect all critical page table entries. Note that we need not copy the entire kernel page table. To understand why this is true, observe that a page table describes a function between virtual and physical addresses. Therefore, given virtual address corresponds to exactly one entry in a page table at each level of the page table hierarchy. Copying and protecting these page tables is sufficient to protect the critical page table entries.

For the Linux kernel, entry pointers point into the main kernel code segment. The main kernel code segment is mapped using pages that are 2MB or 4MB in size, depending upon whether or not PAE mode is being used. In non-PAE mode, the CPU uses a two level page table hierarchy. A 4MB page is mapped by a single entry in the first level page table. Therefore, in non-PAE mode the amount of data that needs to be copied into the shadow table area to protect the critical page table entries reduces to the first level page table (4K) and one second level page table that translates the virtual addresses of the shadow table area (4K). Then the total amount of data copied in non-PAE mode comes to 8K. Note that the second number assumes that the shadow table area, which is 256K in size, will be mapped by one second level page table. In non-PAE mode paging a second level page table can map up to 4MB of memory. As far as the start address of the shadow table area is aligned so that it occupies only addresses translated entirely by one second level page table, copying one page table at the second level should be sufficient. The Linux kernel allocates memory for the shadow table area with this restriction in mind.

In PAE-mode, the CPU uses a three level page table hierarchy. The main Linux kernel mode segment is mapped using a 2MB page. This mapping uses two first two levels of page tables. The shadow table area uses all three levels of page tables. Then, in worst case, the amount of data to be copied to protect the critical page table entries in PAE-mode is the first level page table (32 bytes) and the second level page table (4K) for the entry pointers, and one second level page table and one third level page table (4K). The worst case total in this case is 12K + 32 bytes. Note that as with the non-PAE case this assumes that the shadow table area will be mapped by a single third level page table. Since a third level page table in PAE mode can map up to 2MB of memory this assumption is justified.

In summary, to protect the critical page table entries in the kernel's page tables, SecVisor copies the relevant page tables in all levels of the page table hierarchy into the shadow table area. The guest physical addresses of the shadow table area are marked read-only in the NPT and the system physical pages of the shadow table area are protected from DMA writes. SecVisor also modifies the guest's `cr3` for user mode execution to point to the top-level page table in the shadow table area and modifies the pointers in the page table entries at all levels of the page table hierarchy to to use the copies in the shadow table area. Note that, only a few pointers per level of the page table hierarchy need to be modified.

Now that it is guaranteed that the kernel's page table will correctly the virtual addresses of the shadow table area and the virtual addresses of the entry pointers to guest physical addresses of the shadow table area and the guest physical addresses of the approved code pages respectively, and the guest physical addresses of the shadow table area are marked read-only in the NPT, and the system physical pages of the shadow table area are protected against DMA writes, the shadow GDT, LDT, and IDT cannot be modified during user mode execution. Recall that SecVisor's lazy synchronization code sets the `gdtr`, `ldtr`, and `idtr` to the virtual addresses of the shadow tables and that the MSR-based entry pointers contain the correct virtual addresses. This means that all entry pointers will correctly refer to physical pages containing approved code during user mode execution, thereby satisfying property P1.

# 6. PORTING THE LINUX KERNEL

In this section we discuss how we port the Linux kernel to SecVisor, by illustrating how SecVisor handles the two kinds of code that can be loaded into kernel memory: the main Linux kernel which is loaded at bootstrap, and the kernel modules which are dynamically loaded and unloaded during the lifetime of the system.

## 6.1 Main kernel

The main kernel makes calls to the Basic Input and Output System (BIOS) as part of its initialization. Since the BIOS executes in Ring 0, SecVisor would have approve the BIOS code. However, approving the BIOS code is not simple because of the diversity and complexity of the BIOS subsystem. For example, some BIOS only map a part of the Flash chip containing the BIOS image into the physical address space of the CPU. Others map their code into several different regions of physical memory whose locations might differ from system to system. Approving the BIOS could add considerably to the code size and complexity of SecVisor.

Fortunately, the Linux kernel is designed so that after its initialization, it does not make any more calls to the BIOS. Even more conveniently, the main kernel's code is divided into two parts: the bootstrap part which contains all the calls to the BIOS and the runtime. In view of this, all we need to do is launch SecVisor after the bootstrap finishes execution and SecVisor does not have to deal

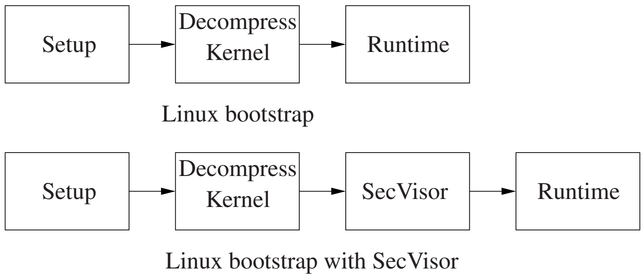


**Figure 9: Modifying Linux's bootstrap. decompress_kernel invokes SecVisor using `skinit`.**

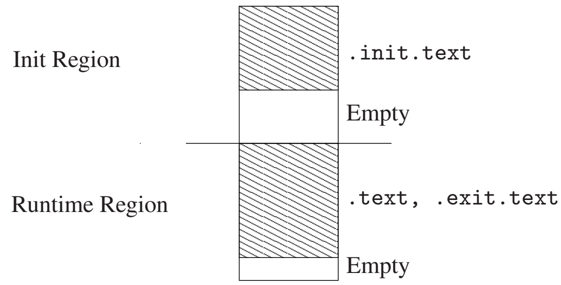


**Figure 10: Layout of different modules sections in memory. In the figure we assume that the Init Region and Runtime Region are both less than one page in size. Both regions start at page aligned addresses and there is empty space between the end of the code and the end of the page.**

with the BIOS at all![4]

Figure 9 shows the normal bootstrap sequence of Linux. The bootloader loads the kernel into memory and jumps to the kernel's bootstrap code. The kernel's bootstrap code consists of two parts: a `setup` function and a `decompress_kernel` function. The `setup` function executes first and initializes the hardware with calls to the BIOS. It then jumps to the `decompress_kernel` function, which performs further hardware initialization, decompresses the runtime, and jumps to start address of the runtime.

We modify this boot sequence to make `decompress_kernel` invoke SecVisor via the `skinit` instruction (the bootloader loads SecVisor into memory along with the kernel) as shown in Figure 9. The late launch feature of `skinit` ensures that SecVisor will now execute untampered by any entity on the system. `decompress_kernel` also passes the start and end addresses of the runtime's code segment as parameters to SecVisor. SecVisor then performs its initialization and passes the runtime image to the approval policy for approval. In our current implementation, we use an approval policy based on a whitelist of cryptographic hashes. Our approval policy computes a SHA-1 [12] hash of the kernel runtime, and approves the runtime if the hash exists in the whitelist. Upon approval, SecVisor creates a VMCB whose CPU state is set to the state of the CPU at the time when the runtime starts executing during a normal bootstrap. Finally, SecVisor sets memory protections over the runtime code, and transfers control to the runtime using the `vmrun` instruction.

We need to address a few additional issues in the above implementation. One is the issue of validating the start and end addresses of the main kernel code image passed by the `decompress_kernel` function. This can be simply handled by noting that if the start and end addresses passed to SecVisor are different from their correct values, then the main kernel code image should differ from its expected value, and should be rejected by the approval policy.

The second issue is that it is impossible, under the W $\oplus$ X protection scheme, to set suitable protections for pages that contain both code and data. To address this issue, we modify the kernel linker script to ensure that start addresses of all data segments of the runtime are page aligned.

## 6.2 Kernel modules

The main issue with modules is that the module loading code in the kernel relocates the module executable to link it with the kernel. The module image will look different after the relocation than before. Since the load address of a module can vary each time it is loaded and can vary between systems, it is impractical to create an approval policy to deal with all possible load addresses of all possible modules on every system. It is also not safe to approve the module code image before relocation by the kernel. Now the kernel will modify the module code image after approval and it is difficult to verify that the kernel's writes are not malicious.

Our solution to the above conundrum is to have SecVisor perform the relocation after subjecting the module to approval by the approval policy. The kernel informs SecVisor of the virtual addresses from which the module will execute. As we show in Section 7, the increase in SecVisor's code size due to the addition of the relocation code is small.

Kernel modules can be loaded and unloaded dynamically. Therefore, SecVisor needs to set protections over module code on a module load and remove these protections on a module unload.

**Module loading.** Linux kernel module files on disk are relocatable object files in the Executable and Linkable Format (ELF) format. All module code is contained in three ELF sections: `.text`, `.init.text`, and `.exit.text`. From the kernel source code, we find that it puts `.init.text` in one memory region and `.text` and `.exit.text` contiguously in another memory region. For the purposes of this description, we call the first memory region the *init region* and the second memory region the *runtime region*. Figure 10 shows how the module code is laid out. The figure shows the two regions as being contiguous pages in memory but this need not always be true. Finally, the kernel relocates the module code using the addresses of the two regions as inputs.

We modify the control flow of the kernel's module loading code in the function `load_module`, so that it invokes SecVisor via a hypercall, after copying the module's code into the init and runtime regions. The arguments to the hypercall are the start and end addresses of the init and runtime regions (virtual addresses in shadow paging, and virtual and guest physical addresses in nested paging). On receiving the hypercall, SecVisor first calls the approval policy to check the module code. As with the main kernel any incorrect arguments to the hypercall will cause the approval check to fail. If the check passes, SecVisor relocates the module based on the arguments of the hypercall. Finally, SecVisor fills the empty space in the init and runtime regions with `no-op` instructions, sets memory protections over the two regions and returns.

**Module unloading.** Unloading a module allows the kernel to reclaim the memory used. We modify the `free_module` function in the kernel to invoke SecVisor via a hypercall. SecVisor makes the pages occupied by the code pages of the module writable by the CPU and peripheral devices and removes their execute permission.

[4] This is not the whole story. A System Management Interrupt (SMI) could still invoke the BIOS. However, SVM provides a way to invoke the System Management Mode (SMM) handler under the control of the host. A description of how we can use this feature of SVM to safely handle SMI is given in Section 8.

| | Debug | Initialization | | Header Files | | Runtime | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPT | NPT | Code | Declarations | SPT | NPT | SHA-1 | Module Reloc |
| Lines of code | 469 | C 538 | C 599 | 376 | 922 | C 1236 | C 609 | 294 | 81 |
| | | Asm 130 | Asm 130 | | | Asm 46 | Asm 46 | | |

**Table 1: Lines of code in SecVisor. SPT and NPT stand for the shadow paging based and nested paging based implementations respectively. For parts of the code that are a mix of C and assembly, we report the counts separately.**

This prevents any further execution of the module code.

# 7. EVALUATION

In this section we evaluate our SecVisor prototype using two metrics: compliance with design requirements and performance.

## 7.1 Design Requirements Compliance

As we mentioned in Section 1 we have three design goals for SecVisor: (1) small code size, (2) minimal kernel interface, and (3) ease of porting OS kernels. The first two goals aid in achieving better security, and the third goal simplifies deployment. We now discuss how our prototype complies with our design goals.

**Code size.** We use D.A. Wheeler's `sloc` program to count the number of lines of source code in our SecVisor prototype. The results are presented in Table 1. For the purpose of measurement, we divide SecVisor's code into four parts. The initialization code initializes the CPU state and SecVisor's runtime state after SecVisor is invoked by the `decompress_kernel` function using the `skinit` instruction. The memory occupied by this code is made available for use by the kernel once the SecVisor runtime code gains control. The debug code provides a `printf` function, which is not required on a production system. The C language header files have both declarations as well as code in the form of preprocessor macros and functions. Finally, the runtime code is responsible for providing the guarantee of approved kernel code execution. We report the code sizes for shadow paging and nested paging implementations separately. Also shown in Table 1 are the code sizes of the SHA-1 function SecVisor uses for its approval policy, and the ELF relocation code for the kernel modules.

As can be observed from Table 1, the total size of the nested paging implementation of SecVisor is 3526 lines of C and assembler code. Of this, the security-sensitive runtime code and the header files measure 2328 lines. When the declarations (which mainly consist of various constants and large `struct` declarations) in the header files are removed from the previous count, the code size comes out to 1406 lines. For the shadow paging implementation, the total code size is 4092 lines of C and assembler code, with the security-sensitive runtime code and header files measuring 2955 lines. Upon removing the declarations from the count we are left with 2033 lines of code. These code sizes should put SecVisor within the reach of formal verification and manual audit techniques.

**Kernel interface.** SecVisor's interface to the kernel consists of only 2 hypercalls. The first hypercall is used by the kernel to request changes to its code (such as loading and unloading modules), while the second hypercall is used by the kernel during its initialization to pass the virtual and guest physical addresses of the shadow table area. The hypercall interface is small which reduces the attack surface available to the attacker through the kernel. Also, the parameters passed in each hypercall are well-defined, making it possible for SecVisor to ensure the validity of these arguments.

**Effort required to port a kernel.** SecVisor's design makes very few assumptions about the kernel which it protects. We now enumerate the changes we had to make to the Linux kernel to port it to SecVisor. Then we point out the kernel specific assumptions that SecVisor makes and discuss how those assumptions affect the effort required to port a new kernel.

We made three changes to the Linux kernel version 2.6.20 to port it to SecVisor. First, the `decompress_kernel` function invokes SecVisor using the `skinit` instruction instead of jumping to the decompressed kernel. Second, during its initialization, the kernel passes the addresses of the shadow table area to SecVisor using a hypercall. Finally, we changed the control flow of the `load_module` and the `free_module` function. As part of changing the control flow of these functions, we removed the ELF relocation code from the `load_module` function and added hypercalls to both functions. In all, the three changes added a total of 12 lines of code to the kernel and deleted 81.

SecVisor makes three assumptions about the kernel it protects. First, it assumes that the user and kernel mode share address spaces. If a kernel design uses separate address spaces for user and kernel modes, then the design of the shadow paging and nested paging code in SecVisor would need to be adapted. However, the changes are relatively small since we would only need to maintain separate page tables for user mode and kernel mode, and handle the page faults that arise when the kernel tries to access user memory. Second, SecVisor assumes that the kernel's binary does not have pages that contain both code and data. Even if a kernel binary does not satisfy this requirement, it should be relatively easy to fix by appropriately modifying the linking of the kernel. Third, in order to not deal with the BIOS, SecVisor requires that the kernel not make any BIOS calls after its initialization. Kernels that do not satisfy this assumption will be relatively difficult to port to SecVisor without adding support in SecVisor for dealing with the BIOS.

## 7.2 Performance Measurements

We now report the performance of the SPT-based SecVisor implementation and compare it to Xen and the Linux kernel. We cannot evaluate the NPT version of SecVisor on real hardware since suitable CPUs are not available at this time. While we have implemented and tested the NPT version of SecVisor using AMD's SimNow simulator, we do not report performance measurements since SimNow is not a cycle accurate simulator.

**Experiment setup.** Our experimental platform is the HP Compaq dc5750 Microtower PC. This PC uses an AMD Athlon64 X2 dual-core CPU running at 2200 MHz, and has 2 GB RAM. SecVisor allocates 1536 MB of RAM to the kernel in our experiments. The PC runs the i386 version of the Fedora Core 6 Linux distribution. We use the uniprocessor versions of Linux kernel 2.6.20 and Xen 3.0.4. All our Xen experiments execute in dom0. Our experiments consist of kernel microbenchmarks and application benchmarks.

**lmbench microbenchmarks.** We use the `lmbench` benchmarking suite [15] to measure overheads of different kernel operations when using SecVisor. SecVisor adds overhead to kernel operations in three ways: (1) by modifying execute permissions in the SPT on each transition between user and kernel mode, (2) by synchro-

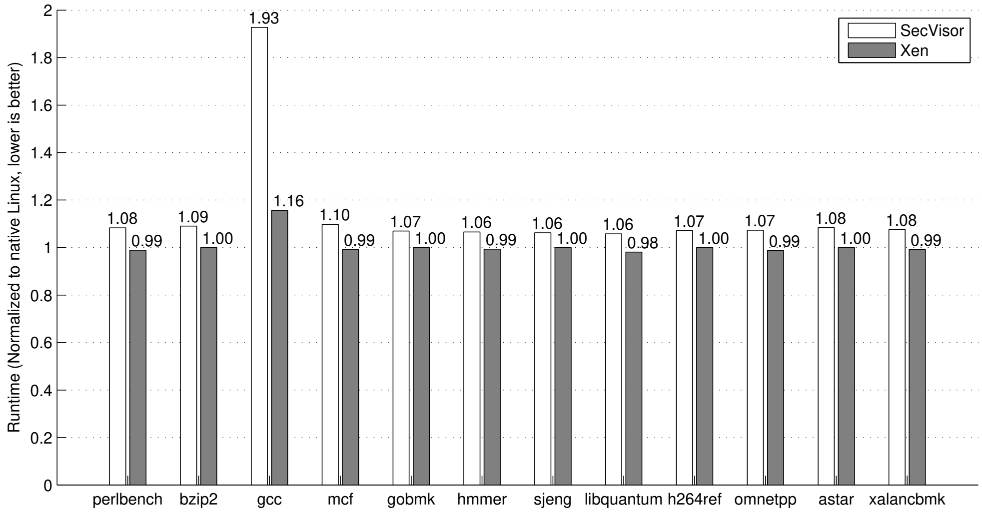


**Figure 11: SPECint 2006 performance comparison between SecVisor and Xen, relative to native Linux.**

nizing the SPT with the kernel's page table, and (3) by shadowing the GDT, LDT, and IDT. We use a subset of the process, memory, and context switch microbenchmarks from `lmbench` to study these overheads.

| Host | Null Call | Fork | Exec | Prot Fault | PF |
|---|---|---|---|---|---|
| Linux (UP) | 0.10 | 139 | 410 | 0.248 | 1.71 |
| Xen (UP) | 0.17 | 415 | 1047 | 0.565 | 3.71 |
| SecVisor | 25.6 | 2274 | 6203 | 27.3 | 35.1 |

**Table 2: Execution times of `lmbench` process and memory microbenchmarks. All times are in $\mu$s. PF stands for page fault and UP for uniprocessor.**

Table 2 shows the results of our experiments. The `Null Call` shows the overhead of a round trip between user and kernel mode and back, i.e., it shows the overhead of (1) and (3) above. The `Prot Fault` indicates the time taken by the kernel to process a write access violation. The overhead is quite close to that of `Null Call` since it also only involves a round trip from user to kernel mode and back, from the perspective of SecVisor. The overhead of `Page Fault` is higher than that of `Prot Fault` since handling a page fault requires a round trip from user mode to kernel mode and back, in which the kernel updates its page table, followed by a round trip from user mode to SecVisor and back, in which SecVisor synchronizes the SPT with the kernel's page table. The `Fork` and `Exec` microbenchmarks incur all three sources of overhead.

| Source | Null Call | Fork | Exec | Prot Fault | PF |
|---|---|---|---|---|---|
| SPT | 0.10 | 1275 | 3043 | 2.289 | 14.6 |
| SPT + Perm | 21.8 | 2148 | 5816 | 22.5 | 32.9 |

**Table 3: Breakdown of the SecVisor overheads in the `lmbench` process and memory microbenchmarks. All times are in $\mu$s. PF stands for Page Fault, SPT for shadow page tables, Perm for modifying execute permissions on user and kernel memory.**

In order to obtain an understanding of how much each of the three sources of overhead contribute to the overall overhead, we conduct further experiments. For these experiments, we implement two additional versions of SecVisor: one that only virtualizes physical memory using the SPT, and the other that modifies the execute permissions of user and kernel memory in addition to virtualizing physical memory. These implementations allow us to isolate the overheads of the three sources.

Table 3 shows our results. From comparing the first and second rows of this table it is clear that modifying the execute permissions for user and kernel memory drastically increases the overhead in all benchmarks. Also, by comparing the last row of Table 3 with the last row of Table 2 is it obvious that shadowing the GDT, LDT, and IDT is a much lower overhead operation than modifying the execute permissions.

| Host | 2p/0K | 2p/16K | 2p/64K | 8p/16K | 8p/64K |
|---|---|---|---|---|---|
| Linux (UP) | 0.56 | 0.64 | 3.19 | 1.48 | 12.9 |
| Xen (UP) | 2.61 | 2.42 | 5.16 | 4.07 | 17.1 |
| SecVisor | 54.3 | 52.7 | 53.6 | 63.3 | 75.8 |

**Table 4: Execution times of `lmbench` context switch microbenchmarks. All times are in $\mu$s. UP stands for uniprocessor.**

Table 4 shows the results of running `lmbench` context switch microbenchmarks. The context switch incurs the overhead of all three sources, leading to significant slowdown in SecVisor compared to the native Linux kernel and Xen.

**Application benchmarks.** We hypothesize that when SecVisor is used, the overhead of an application will be directly proportional to both the number of times the application calls the kernel and the rate of change of the application's working set. Kernel calls and returns will impose the overhead of switching execute permissions in the SPT and shadowing the GDT, LDT, and IDT, while a change of the working set will impose the overhead of SPT synchronization. Based on our hypothesis, compute-bound applications that have a stable working set throughout their lifetime will have the lowest overhead. On the other hand, I/O bound applications with highly variable working sets will be the pathological cases.

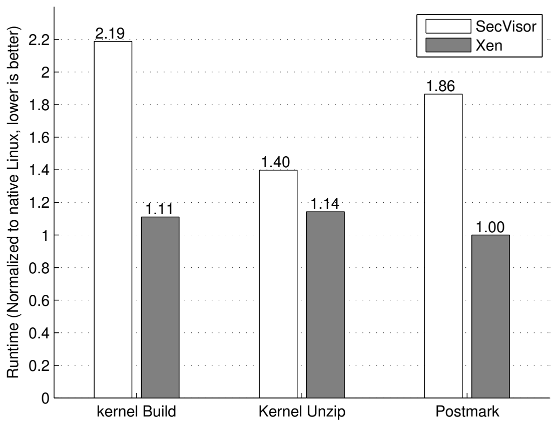


**Figure 12: Application performance comparison between SecVisor and Xen, relative to native Linux.**

To test our hypothesis, we execute both compute-bound and I/O bound applications with SecVisor. For our compute-bound applications we choose benchmarks from the SPECint 2006 suite. Our I/O bound applications consist of the gcc benchmark from SPECint 2006, the Linux kernel compile, unzipping and untarring the Linux kernel sources, and Postmark.

In the Linux kernel compile, we compile the sources of the kernel version 2.6.20 by executing "make" in the top-level source directory. For unzipping and untarring the kernel source we execute "tar xfvz" on the source tarball of the version 2.6.20 of the Linux kernel. For Postmark, we choose 20000 files, 100000 transactions, and 100 subdirectories, and all other parameters are set at their default values. For comparison purposes, we also execute SPECint 2006 and each of these applications on the native Linux kernel and on Xen. We run each of these applications five times on each of the Linux kernel, Xen, and SecVisor.

Our results are presented in Figure 11 and Figure 12. The results are in line with our predictions: the compute-bound SPEC benchmarks have a low overhead while the gcc SPEC benchmark, kernel compile, and Postmark which are I/O bound and have rapidly changing working sets have the highest overheads.

# 8. LIMITATIONS AND FUTURE WORK

We now discuss the limitations of SecVisor, our future implementation plans, and two additional applications of SecVisor.

## 8.1 Limitations

SecVisor guarantees integrity of the code that executes in kernel mode, but not the integrity of the control flow. Consequently, a "return-to-libc" style attack within the kernel is possible, where an attacker can cause a `return` instruction in the kernel code to pass control to a kernel function of its choosing by overwriting the return address stored on the stack. Another attack that manipulates the control flow of a program is described by Shacham. This attack constructs arbitrary instruction sequences by adequately combining existing instruction sequences [21]. However, such attacks can be mitigated by combining SecVisor with techniques that guarantee control flow integrity [6, 14].

Besides manipulating the control flow to perform attacks, the attacker could also modify kernel data in order to indirectly influence the control flow within kernel code. For example, the attacker could change the predicate of a conditional. Attacks of this kind are discussed by Chen et al. in the context of user applications [4].

## 8.2 Future Implementation

We now present our future extensions to the SecVisor prototype, presented in their expected order of their implementation.

**Multi-CPU support.** With the trend towards multi-core CPUs, it becomes necessary for SecVisor to support multiple CPUs. Implementing this support requires SecVisor to intercept the attempt by the kernel's initialization code to switch from uniprocessor to multiprocessor operation. Also, SecVisor needs to implement locking primitives to synchronize access to its global data. Finally, SecVisor needs to set up CPU data structures such as the GDT, IDT, and VMCB for each CPU in the system.

On a multi-CPU AMD x86 system, one CPU is elected as the bootstrap processor (BSP) each time the system is powered up. All other CPUs become application processors (AP). The `skinit` instruction can only be executed by the BSP. All the APs must be put into a halted state by the code that invokes `skinit`. The APs will remain in the halted state until they are woken up by code executing on the BSP using an interprocessor interrupt (IPI).

The `decompress_kernel` function starts SecVisor by invoking `skinit`. SecVisor then executes on the BSP, and starts the kernel's runtime by executing the `vmrun` instruction. The kernel's runtime initially executes on the BSP. Sometime during its initialization, the kernel switches to multiprocessor mode by waking up all the APs via an IPI. SecVisor can intercept the kernel's attempt to send this IPI, and perform the AP wake up. Then it can set up a VMCB for each AP, and execute the `vmrun` instruction on each of them. This causes all the APs to begin executing the kernel's runtime, thereby simulating the effect of the kernel's IPI.

SecVisor's locking primitives can be simple since the amount of global state in SecVisor is small. We plan to use *spinlocks* as SecVisor's locking primitive.

**Handling SMI.** The System Management Mode (SMM) is a special operating mode of an x86 CPU that is entered when the CPU receives a System Management Interrupt (SMI). SMM is designed to be transparent to all software executing on the system. It is normally used by the BIOS to perform power management tasks or to fix hardware bugs. For example, closing the lid of a laptop generates an SMI that causes the SMM handler in the BIOS (in conjunction with the OS) to suspend the laptop.

The SMM handler executes at the highest CPU privilege so that it can perform its tasks transparently to the rest of the system. This is not desirable from SecVisor's point of view since it now has to trust the BIOS code. However, SVM provides a way to address this issue. SecVisor can intercept the SMI and execute the SMM handler in guest mode. Then SecVisor can prevent the SMM handler from modifying CPU and memory state that are necessary to guarantee kernel code integrity. We expect that this restriction will not affect the execution of the SMM handler since SecVisor will only prevent the SMM handler from modifying memory pages of SecVisor and those of the kernel code, the MSRs containing entry pointers, and the DEV mechanism configuration registers. Another option is to use a BIOS without SMM code, such as LinuxBIOS [16].

**Self-modifying code.** SecVisor can detect the kernel's use of self-modifying code through write faults. When such a write fault occurs SecVisor can perform the write on behalf of the kernel. SecVisor then calls the approval policy to approve the modification to the kernel code. Approving the write requires determining if the write is genuine or was initiated by an attacker. The difficulty can be somewhat mitigated, if the approval policy is aware of the reason for the code alteration. For example, a kernel might fix compiler or CPU bugs by modifying its code at bootstrap. Since the list of such bugs is well known, the approval policy can check if the modification being performed to the kernel code is a bug fix. In general, however, supporting self-modifying code is likely to complicate the approval policy of SecVisor.

**Porting to Intel TXT and Windows XP.** Intel's Trusted Execution Technology (TXT) is a CPU-based virtualization and security technology present in recent CPUs from Intel [10]. It provides facilities that are semantically similar to those of AMD SVM. Therefore it should be possible to port SecVisor to systems with TXT support. We also plan to port the Windows XP kernel to SecVisor.

### 8.3 Additional Applications

**Protecting user programs.** SecVisor should be naturally applicable to protecting the code of user programs as well. Preventing code injection attacks against user programs should help mitigate the threat from several current generation worms that use code injection as their method of attack. Also, it can prevent attackers from creating bots by exploiting vulnerabilities in user programs to inject the bot code. However, the overhead of providing user code protection in SecVisor is likely to be higher than that of protecting kernel code since allocation of user memory pages between code and data changes more often (for example, on every context switch).

**Kernel code attestation.** SecVisor can implement an approval policy for kernel code attestation. Such an approval policy can compute and store cryptographic hashes of all code that is loaded into kernel memory from the time the kernel is started. The attestation offered by SecVisor does not suffer from a time-of-check-to-time-of-use (TOCTTOU) problem since SecVisor will not permit kernel code to be modified without re-hashing the new code.

## 9. RELATED WORK

In this section, we survey proposed techniques for ensuring kernel code integrity, small virtual machine monitors, and kernel rootkit detection.

### 9.1 Kernel Code Integrity Protection

The common approach followed by all mainstream OSes to ensure kernel code integrity is through kernel access control mechanisms, such as file system protections (to prevent unauthorized alterations of kernel and module binaries, and configuration files) and kernel-enforced restrictions on module loading. Unfortunately, these approaches cannot protect against kernel vulnerabilities.

The IBM 4758 secure coprocessor provides special hardware support for a "ratchet" mechanism, which locks OS memory and loaded modules after the ratchet mechanism has been incremented [7, 22]. The 4758 hardware support offers a high level of security thanks to the hardware-enforced mechanism, but is inflexible since it prevents dynamic changes the kernel code, and would not be applicable to mainstream OSes.

Program Shepherding attempts to provide code integrity and control flow integrity for application programs [14]. It uses a dynamic optimization framework to check that every control transfer in the program satisfies the specified security policy and also checks the origin to the program's executable code. SecVisor can be used to protect the dynamic optimization framework used by Program Shepherding, thereby achieving stronger security guarantees.

Livewire is a host-based Intrusion Detection System (IDS) built into a VMM [9]. It detects intrusions by observing the state of the kernel executing in a VM. To prevent an attacker from injecting malicious code into the kernel to manipulate its state, Livewire uses the VMM to make the kernel code segments read-only. However, since Livewire does not address properties P1 and P3, there is no guarantee that the CPU will not execute code outside the kernel's code segments in kernel mode.

Recently, Criswell et al. have proposed the Secure Virtual Architecture (SVA), which uses programming language techniques to provide memory safety and control-flow integrity for commodity kernels [6]. Analogous with Program Shepherding, SVA can be combined with SecVisor to achieve stronger security guarantees by using SecVisor to protect SVA's runtime environment.

### 9.2 Small Virtual Machine Monitors

There have been several efforts to build small VMMs [8, 13, 18]. The aim of these VMMs is to minimize code size until it is small enough for formal verification or manual audit. Such VMMs could be adapted to provide properties similar to SecVisor.

However, the security properties of SecVisor will be better than those of VMMs because of two reasons. First, the code size of SecVisor will be smaller than that of a full VMM for two reasons. One, SecVisor only virtualizes the MMU, the IOMMU, and the physical memory whereas a VMM has to virtualize the entire system. Two, unlike a VMM, SecVisor does not need to support multiple Virtual Machines (VM). This reduces the code size by eliminating certain features such as context switching, scheduling, and interrupt handling. Also, the memory, MMU and IOMMU virtualization code of SecVisor will be smaller than the corresponding code in a VMM. Second, in addition to code size, the size of SecVisor's external interface will also be smaller than that of a VMM (for example, VMMs need to provide interfaces for administering VMs).

### 9.3 Kernel Rootkit Detection

Various techniques have been proposed to detect malicious code in the OS kernel (also called rootkits). As far as we are aware, SecVisor is the only technique that provides the stronger property of *preventing* code injection attacks against OS kernels. Rootkit detection techniques can be classified into two categories: software-based and hardware-based. Attacks exist against both hardware and software-based rootkit detection techniques.

**Software-based kernel rootkit detection.** Rootkit detection techniques that rely on the integrity of one or more parts of the kernel can be defeated by an attacker that compromises the parts of the kernel whose integrity is relied on by the rootkit detector [5, 11, 24]. Even rootkit detectors that do not rely on the integrity of any software executing on a machine can be defeated by placing the rootkit in the OS's data segment or heap whose integrity is difficult to check since it is not practical to determine in advance what the "known good" value of dynamic segments of memory will be [20]. Rootkit detection techniques that rely on differences in file system scans cannot detect rootkits that do not modify the file system [25].

**Hardware-based rootkit detection.** Hardware-based rootkit detection techniques work by attaching specialized peripheral devices to a system. These devices check the integrity of one or more regions of the kernel memory to detect the presence of malicious code. Recently, Rutkowska demonstrated a generic attack against hardware-based rootkit detectors that relies on the dichotomy of the CPU's view of physical memory and the devices' view of physical memory [19]. We mention a generic version of the same attack against Copilot [17] in our earlier work [20].

## 10. CONCLUSION

With the general observation that the number of security vulnerabilities increases exponentially with complexity and size, it is no surprise that critical vulnerabilities are frequently discovered in mainstream kernels.

In this context, we asked: what is the minimal change we can introduce to significantly enhance the security of legacy kernels? Leveraging the features of the next generation of commodity CPUs, we design SecVisor, a tiny hypervisor that protects the code integrity of kernels during the system lifetime, and ensures that only approved code can execute in kernel mode. SecVisor protects the kernel against a variety of well-known and unpublished attacks, including code injection through buffer overruns, kernel-level rootkits, and malicious devices with DMA access. So far, the majority of approaches to secure OSes follow a *detection* approach, which detect and mitigate attacks. SecVisor takes a more direct approach, which is to *prevent* a large class of attacks altogether.

While SecVisor does not prevent against control-flow attacks, it can be combined with approaches that do provide additional protections. Moreover, SecVisor will ensure code integrity and memory protection for such additional security mechanisms.

## 11. ACKNOWLEDGEMENTS

We gratefully acknowledge the help provided by AMD in donating suitable hardware and providing technical support. We would also like to thank our shepherd Richard Draves, as well as Benjamin Serebrin, Leendert van Doorn, Elsie Wahlig, Daniel Wendlandt, and the anonymous reviewers for their help and feedback.

## 12. REFERENCES

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.12 edition, September 2006.

[2] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 3.12 edition, September 2006.

[3] M. Becher, M. Dornseif, and C.N. Klein. FireWire all your memory are belong to us. In *Proceedings of CanSecWest*, 2005.

[4] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R.K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, pages 177–192, August 2005.

[5] A. Chuvakin. Ups and downs of UNIX/Linux host-based security solutions. *;login: The Magazine of USENIX and SAGE*, 28(2), April 2003.

[6] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of ACM Symposium on Operating Systems Principles*, Oct 2007.

[7] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[8] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *In Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[9] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[10] Intel Corporation. Trusted eXecution Technology – preliminary architecture specification and enabling considerations. Document number 31516803, November 2006.

[11] K. J. Jones. Loadable Kernel Modules. *;login: The Magazine of USENIX and SAGE*, 26(7), November 2001.

[12] P. Jones. RFC3174: US Secure Hash Algorithm 1 (SHA-1). `http://www.faqs.org/rfcs/rfc3174.html`, September 2001.

[13] K. Kaneda. Tiny virtual machine monitor. `http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/`.

[14] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[15] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, Jan 1996.

[16] R. Minnich, J. Hendricks, and D. Webster. The Linux BIOS. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Oct 2000.

[17] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of USENIX Security Symposium*, pages 179–194, 2004.

[18] R. Russell. Lguest: The simple x86 hypervisor. `http://lguest.ozlabs.org/`.

[19] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. In *Proceedings of BlackHat DC 2007*, Feb 2007.

[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, October 2005.

[21] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Oct 2007.

[22] S.W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31:831–960, 1999.

[23] J. von Neumann. First draft of a report on the EDVAC. In B. Randall, editor, *The origins of digital computers: selected papers*, pages 383–392. 1982.

[24] Y. Wang, R. Roussev, C. Verbowski, A. Johnson, and D. Ladd. AskStrider: What has changed on my machine lately? Technical Report MSR-TR-2004-03, Microsoft Research, 2004.

[25] Y. Wang, B. Vo, R. Roussev, C. Verbowski, and A. Johnson. Strider GhostBuster: Why it's a bad idea for stealth software to hide files. Technical Report MSR-TR-2004-71, Microsoft Research, 2004.