Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor

Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, Anupam Datta

June 16, 2008
CMU-CyLab-08-008

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor *

Jason Franklin, Arvind Seshadri, Ning Qu
Carnegie Mellon University

Sagar Chaki
Software Engineering Institute

Anupam Datta
Carnegie Mellon University

Abstract

SecVisor is a hypervisor designed to guarantee that only code approved by the user of a system executes at the privilege level of the OS kernel [17]. We employ a model checker to verify the design properties of SecVisor and identify two design-level attacks that violate SecVisor's security requirements. Despite SecVisor's narrow interface and tiny code size, our attacks were overlooked in both SecVisor's design and implementation. Our attacks exploit weaknesses in SecVisor's memory protections. We demonstrate that our attacks are realistic by crafting exploits for an implementation of SecVisor and successfully performing two attacks against a SecVisor-protected Linux kernel. To repair SecVisor, we design and implement an efficient and secure memory protection scheme. We formally verify the security of our scheme. We demonstrate that the performance impact of our proposed defense is negligible and that our exploits are no longer effective against the repaired implementation. Based on this case study, we identify facets of secure system design that aid the verification process.

1 Introduction

Operating system kernels in common use are becoming increasingly complex in order to support new hardware and applications. The complexity of the kernel leads to security vulnerabilities in its design and implementation. Since kernel code typically executes at the highest privilege level, an attacker can gain complete control of the system by exploiting such a vulnerability. A spate of recent research in the OS and system security communities seeks to address this important problem by incorporating various security mechanisms into newly designed kernels [5, 19, 22] or by using additional components such as hypervisors to provide security guarantees even if the kernel is compromised [20, 3].

Issues involved in the design, implementation, and performance of such systems have received considerable research attention. However, a critical missing piece in many cases is the absence of formal assurance that these systems actually provide their desired security properties. Such assurance is important since secure system designs are complex and security properties have to be guaranteed even in the presence of concurrently executing attackers who actively try to subvert the system.

Recently, a subset of the co-authors (along with collaborators) designed and implemented a security hypervisor[1] called SecVisor [17]. SecVisor's design goal is to guarantee that only user-approved code executes in the OS kernel mode. It's Trusted Computing Base (TCB) consists of only the CPU, memory controller and system memory. SecVisor aims to guarantee its design objective against an attacker who has complete control over the rest of the system; in particular, the attacker can compromise the kernel.

The design and implementation considerations of SecVisor have been presented previously [17]. In this paper, we fill in an important missing piece: we report on the results of, and lessons learned from, a formal security analysis of SecVisor against its design goal using the Murφ model checker [4]. Our analysis identified serious vulnerabilities in the SecVisor design and implementation. We repaired the design and fixed the implemen-

*This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and MURI W 911 NF 0710287 from the Army Research Office, grant CNS-0347807 from the National Science Foundation, a faculty fellowship from the Sloan Foundation, the Predictable Assembly from Certifiable Components (PACC) initiative at the Software Engineering Institute (SEI), and the NSF Science and Technology Center TRUST. Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow under the DHS Scholarship and Fellowship Program. 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 ARO, CMU, DHS, NSF, SEI, or the U.S. Government or any of its agencies.

[1] A hypervisor is software that executes at a CPU privilege greater than that of the supervisor (the OS kernel).

tation of SecVisor and verified that the resulting design satisfied the desired properties without significant performance loss. During the course of our analysis, we identified three general facets of secure systems design that aid the verification process: (1) clear specification of the adversary model, (2) explicit statement of the top-level security requirements that the system aims to provide, and (3) clean separation between these requirements and the mechanisms used to achieve them. We elaborate further on these facets in Section 6.

Overview of Contributions. *First*, we develop a Murφ model (see Section 3 for details) of the SecVisor design. Our model consists of three parts: (1) the hardware platform (2) a model of SecVisor, and (3) a model of the attacker.

Our model also includes a precise formulation of the security properties as *invariants*[2]. The invariants refer to two distinct models—an *ideal* model and an *actual* model. In the ideal model, the system is executed free from the influences of the adversary. In contrast, the actual model includes an adversary. The invariants express an agreement between the actual and ideal models on the values of certain variables. Our use of two models is inspired by Lie et al.'s use of two worlds in the verification of the XOM architecture [11].

A central challenge in developing this model was to find the right level of abstraction for capturing the essential features of SecVisor, while ensuring verification completes. We focus on SecVisor's software memory virtualization subsystem primarily because it is a security critical interface where data is copied from an untrusted domain under the control of the adversary (the kernel page table) to a trusted domain (the shadow page table maintained by SecVisor).

Second, we automatically verify our SecVisor model using Murφ. During verification, Murφ found executions that violate execution and code integrity. The vulnerabilities result from flaws in SecVisor's software memory virtualization subsystem, specifically, the shadow page table synchronization code[3]. Furthermore, we demonstrate that the vulnerabilities enable an attacker to take control of a SecVisor-protected kernel by implementing *exploits* and launching two successful code injection attacks. The first exploit is 37 lines of C code while the second is 15 lines. Section 4 describes the attacks on the design as well as the code for the exploits.

Finally, to repair SecVisor, we design and implement an efficient and secure memory protection scheme. We incorporate the fix in our SecVisor model and use Murφ to verify that the augmented model is free from vulnerabilities. Furthermore, we incorporate our protection scheme in SecVisor's implementation by modifying 105 lines of C code, and check that the exploits on the original version of SecVisor no longer compromise code or execution integrity. Last, but not least, we demonstrate that our defense is efficient by running benchmarks on the repaired implementation; performance loss due to our memory protection scheme ranges from 1–7%. The repaired design, implementation, verification and performance results are reported in Section 5.

2 SecVisor Overview

In this section we present a brief overview of SecVisor. We first present the assumptions and threat model for SecVisor, and then present SecVisor's conceptual design.

Assumptions. We assume that the CPU on which SecVisor executes has secure virtualization support, similar to AMD's Secure Virtual Machine (SVM) [1] and Intel Trusted Execution Technology (TXT) [9]. We also assume that the kernel executes non-self-modifying code in 32-bit mode on the x86 architecture. Finally, we assume that the CPU has two levels of execution privilege – *user mode* and *kernel mode* – with kernel being the more privileged mode in which the kernel executes.

Threat Model. We consider an attacker who controls everything in the system except its TCB – the CPU, memory controller, and system memory. Also, the attacker is aware of zero-day vulnerabilities in the kernel and application software, and uses these vulnerabilities to locally or remotely exploit the system.

Conceptual Design. Recall that SecVisor aims to ensure that only user-approved code executes in kernel mode. SecVisor uses a user-supplied policy to approve code for execution in kernel mode. Achieving the goal of user-approved code execution in the kernel requires fulfilling two requirements: (1) execution integrity and (2) code integrity. The execution integrity requirement says that the CPU should only execute instructions from memory regions containing approved code while the system is in kernel mode. The code integrity requirement says that approved code in system memory should only be modifiable by SecVisor and SecVisor's TCB. SecVisor aims to achieve these two integrity requirements via a set of four lower-level properties (**P1-P4**) , which we present next.

Execution Integrity. Recall that every CPU has an Instruction Pointer (IP) register containing the address of the next instruction to be fetched for execution. To satisfy the execution integrity requirement we require that when in kernel mode, the IP always point inside memory regions containing approved kernel code. Since the CPU

[2]Technically, an invariant is a condition that holds on all reachable states of a system. Thus, our goal is to verify that the proposed invariants are valid on our model.

[3]These vulnerabilities were identified independently on SecVisor's implementation by two of the co-authors, but not reported in any peer-reviewed publication. However, an informal update to the SecVisor paper [17] detailing the vulnerabilities is available.

transitions between user and kernel mode, the above requirement boils down to the following three properties:

- **P1**: Every entry into kernel mode (which occurs at the instant the privilege of the CPU changes to kernel mode) should set the IP to an address within memory that contains approved code.
- **P2**: The IP should continue to point to memory regions containing approved code as long as the CPU executes in kernel mode.
- **P3**: Every exit from kernel mode (which occurs at the instant the IP is set to an address in user memory) should modify the privilege level of the CPU to user mode.

Code Integrity. To satisfy code integrity, we first examine what entities on a computer system can perform memory writes. Then, we construct SecVisor so that it marks all memory pages containing approved code as read-only to all entities other than itself and its TCB. On any computer system, memory can be written by software executing on the CPU and by DMA writes by peripheral devices. Since all non-SecVisor code and all peripheral devices are outside SecVisor's TCB, the code integrity requirement reduces to the following property:

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

The question then is: how can SecVisor achieve these four properties while only relying on a TCB consisting of the CPU, the memory controller, and the system memory chips? In the rest of this section, we first discuss how SecVisor achieves **P1** through **P3** by controlling all kernel mode entries and exits using a combination of the CPU architecture specification and hardware memory protections, and then discuss how it achieves **P4** using hardware memory protections.

Achieving P1: Kernel Mode Entry. Each control transfer to kernel mode has an associated data structure, such as the Interrupt Vector Table (IVT). The kernel writes the address of the kernel entry point for each control transfer method in the corresponding data structure. Henceforth, we will call these the *entry pointers*. To achieve **P1**, SecVisor ensures that all CPU-architecturally-defined control transfers to kernel mode sets the IP to an address within memory containing approved code. To this end, SecVisor ensures that all entry pointers point to addresses within memory containing approved code.

Achieving P2: Kernel Mode Execution. In order to achieve **P2**, when in kernel mode, SecVisor must give execute permissions to only memory regions containing approved code. However, this is complicated by the fact that in most popular OSes the kernel and user memories share the same address space. It is clear that the user memory must have execute permissions in user mode. Therefore, we require a mechanism to turn off execute permissions for user memory when entering kernel mode. To this end, SecVisor intercepts all transitions from user to kernel mode to modify user memory execute permissions. To intercept all user mode to kernel mode transitions, SecVisor uses the fact that all entry pointers point to memory regions containing approved code (via **P1**). In user mode, SecVisor makes all memory regions containing approved code non-executable. Since every entry pointer points to non-executable memory, the CPU throws an exception as soon as it enters kernel mode. The handler for this exception (which is also part of SecVisor) removes execute permission for user memory and makes approved code regions executable.

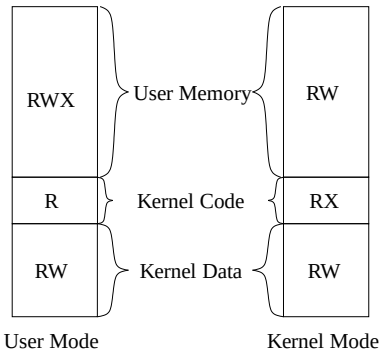

Figure 1: Memory protections for user and kernel modes. R = read, R = write, X = execute. If the kernel's permissions differ from those of SecVisor the actual permissions is the more restrictive of the two. For example, if the kernel marks its data segment read-only in user mode, then, in user mode, the actual permissions for the kernel data segment is R instead of the RW.

Achieving P3: Kernel Mode Exits. To achieve **P3**, SecVisor intercepts all kernel mode exits and sets the privilege level of the CPU to user mode. All legitimate methods for exiting kernel mode transfer control to code in user memory. Recall, from above, that on every kernel mode entry, SecVisor makes user memory non-executable. Then, all kernel exits cause the CPU to try to execute non-executable memory, and throw an exception. The handler for this exception (which is also part of SecVisor) makes memory regions containing approved code non-executable and user memory executable. Note that the mechanisms for achieving **P2** and **P3** rely on each other. Figure 1 shows how SecVisor sets execute permissions for user memory and memory containing approved code during execution in user and kernel modes.

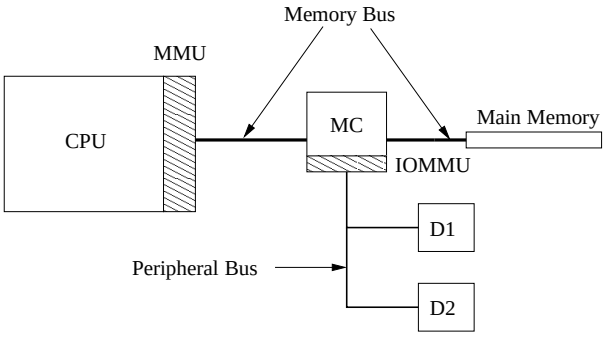

Figure 2: System-level overview of memory protections used by SecVisor. MC is the memory controller, D1 and D2 are peripheral devices. The MMU enforces memory protections for accesses from the CPU while the IOMMU enforces DMA write protections.

Achieving P4: Protecting Integrity of Kernel Code. SecVisor uses hardware memory protections provided by the CPU's Memory Management Unit (MMU) and the memory controller's IO Memory Management Unit (IOMMU) to achieve **P4**. Specifically, SecVisor removes write permissions from memory regions that contain approved code. This prevents any code executing on the CPU (except SecVisor) from writing to these memory regions, thereby satisfying part of property **P4**. 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 ensure that property **P4** is satisfied. Figure 2 shows a system-level overview of the two hardware memory protections used by SecVisor.

3 Modeling SecVisor

In this section, we present an overview of our model for SecVisor, provide background on the Murφ model checker, and detail our formal SecVisor model for Murφ, including the state transition system, adversary model, and security properties.

3.1 Overview of Model

We first present a functional overview of the hardware memory protections used by SecVisor. In this context, we describe the relevant memory protection data structures, the entities that use and manipulate these data structures, and the functional relationship between these entities. After the functional overview, we discuss which parts of SecVisor we choose to incorporate in our formal model and reasons for making these choices.

Hardware Memory Protections. Figure 3 shows the functional overview of the memory protections used by SecVisor. From the description of SecVisor in Section 2, we see that managing hardware memory protections is the principal technique that SecVisor uses to achieve the properties of code integrity and execution integrity. SecVisor sets up suitable data structures for use by the MMU and the IOMMU, and relies on the MMU and IOMMU to enforce the memory protections.

Page tables are the basis of SecVisor's MMU-based hardware memory protections. SecVisor uses a separate set of page tables than the kernel's page tables in order to set its memory protections. To this end, SecVisor virtualizes physical memory, which causes the addresses sent on the memory bus to be different from the physical addresses seen by the kernel.

Hardware and Software Virtualization. Physical memory can be virtualized using either software or hardware virtualization techniques. Software memory virtualization requires SecVisor to maintain a new set of page tables called the Shadow Page Tables (SPT). The SPT translate virtual addresses to the physical addresses sent on the memory bus. SecVisor synchronizes the SPT with the kernel's page tables (KPT) to reflect the changes the kernel makes to the KPT in the SPT. According to our threat model, the adversary is able to read and write the KPT. Thus, synchronization of the SPT with the KPT allows the adversary to indirectly influence the contents of the SPT by altering the KPT. Hardware memory virtualization uses Nested Page Tables (NPT). Unlike the SPT, the NPT do not need to be synchronized with the KPT. This makes the NPT handling code simpler than code that handles the SPT.

The IOMMU-based DMA write protection uses the Device Exclusion Vector (DEV). The DEV is a bit vector with one bit per page of physical memory. Setting the bit corresponding to a page causes the IOMMU to disallow DMA reads and writes to the page from peripheral devices. Unlike the SPT, the DEV is isolated from the adversary and is only accessible to SecVisor.

Modeling Choices. Our formal model of SecVisor encompasses two major modeling choices. First, our model focuses on SecVisor's software memory virtualization subsystem, which includes the management and synchronization of the shadow page tables maintained by SecVisor. We choose to focus on the software memory virtualization for the following reasons:

1. Software memory virtualization is a security-critical interface where data is copied from an untrusted domain under the control of the adversary to a trusted domain.

2. Management and synchronization of the shadow page table requires careful design and exhaustive safety checks; a single flaw is capable of rendering SecVisor vulnerable to attack.

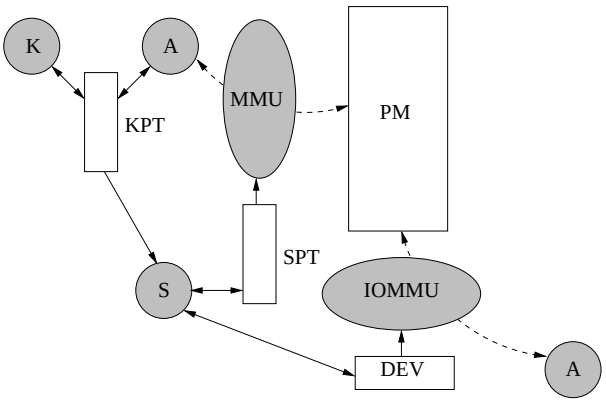

Figure 3: Functional overview of the memory protections used by SecVisor. K is the OS kernel, A the adversary, S is SecVisor, PM is physical memory, KPT are the kernel's page tables, SPT are the shadow page tables, and DEV is the device exclusion vector. The shaded shapes in the picture represent the active elements of the system. The direction of arrows joining the active elements to the data structures indicate the kind of access (read or write) the element performs on that data structure.

3. Our primary focus in this paper is on software design rather than on hardware; in terms of implementation size, the software memory protection runtime is more than double the hardware memory virtualization runtime.

Second, our model includes both of SecVisor's design requirements – execution integrity and code integrity – but to varying degrees. *Execution integrity* requires that the CPU only execute SecVisor approved code in kernel mode. As discussed in Section 2, this requirement translates to properties **P1-P3**. Verifying **P1** and **P3** requires a detailed model of the x86 hardware and, we believe, does not lead to interesting insights into SecVisor's design. In contrast, property **P2** states that if the system is in kernel mode then the instruction pointer must point to approved kernel code. Clearly, this property depends on SecVisor's design and hence is included in our model.

On the other hand, *code integrity* requires that approved code should only be modifiable by SecVisor. SecVisor makes two critical assumptions: (a) memory can only be modified by code executing on the CPU or through Direct Memory Access (DMA) writes by peripheral devices, and (b) kernel code remains static, i.e., self-modifying code is disallowed. Under these two assumptions, we cast code integrity as stating that once code is approved by SecVisor, it cannot be altered during the execution of the system.

3.2 Murφ Background

We use the Murφ model checker to verify SecVisor. Murφ is an explicit-state enumeration tool that searches finite state systems to determine if only "safe" states are reachable. The state space to be searched is determined by a set of states, and rules that specify (possibly non-deterministic) transitions between states. Each transition rule is a guarded command, and consists of a condition (guard) and an action (command). An action, which is executed iff the corresponding guard is satisfied, yields successor states. The set of "safe" states is specified via an *invariant* condition Inv. In other words, a state is "safe" iff it satisfies Inv.

Murφ exhaustively searches the state space, starting from the initial states and continuing until all states reachable via the transition rules have been explored. The invariant specifying the desired safety property is evaluated on every reachable state. If Murφ finds a state s that does not satisfy Inv, it terminates with FAILURE and a counterexample consisting of a finite sequence of states leading from an initial state to s via legal transitions. Otherwise, if all explored states satisfy Inv, Murφ terminates with SUCCESS.

We express security properties as invariants involving two distinct models, an ideal model and an actual model. In the ideal model, the system is free from the influences of the adversary. In contrast, the actual model includes an adversary, and thus deviates from the ideal model. In particular, the ideal model represents the "ground truth" and provides a point of reference with which we compare the actual model to gauge the effectiveness of the adversary. Our use of two models is inspired by Lie et al.'s use of two worlds in the verification of the XOM architecture [11].

3.3 Formal Model

We now present our Murφ model of SecVisor. The variables in the model abstractly represent the hardware platform, memory, and various data structures (such as page tables) which are manipulated by SecVisor, the kernel, and the adversary. The initial states are specified by assigning appropriate values to the model variables. The transition rules of the model correspond to entries into (and exits from) kernel mode, and adversary actions. As mentioned before, these transition rules determine the evolution (and hence the reachable states) of the system. The desired security properties are stated as invariants on the model variables. The overall model consists of around 500 lines of Murφ code.

It is noteworthy that in order to tackle the state space explosion problem our Murφ model bounds the number of memory pages and the size of the page tables. Thus,

our model is an abstraction of the actual SecVisor implementation. In general, if a model M is not a faithful representation of a system S, then the results of model checking on M do not carry over to S. We address this issue in two ways:

1. We model SecVisor conservatively, i.e., our model is an over-approximation of SecVisor in terms of the initial states and transitions (and hence reachable states). Thus, when Murφ terminates on our model with SUCCESS, we conclude that SecVisor satisfies the desired security property as well.

2. Since our model is an over-approximation, counterexamples reported by Murφ on the model may not correspond to actual attacks on SecVisor. To this end, for each security vulnerability of the model identified by Murφ, we construct a corresponding concrete attack against the SecVisor implementation. Further details of this step, with actual exploit code, are presented in Section 4.

3.3.1 Modeling Data

In order to model the hardware platform, memory, and other data structures, we use a number of basic data types including a bit, word, and indexes into the page tables and physical memory, as well as arrays and records. We now present each data type in our model, with a textual description followed by a snippet of Murφ code. All undefined identifiers (e.g., `page_index` and `KRNL_DATA`) are pre-defined integer constants.

Hardware Platform. The hardware platform is modeled as a record data type containing a physical memory, a CPU mode bit corresponding to either kernel or user mode, an instruction pointer (program counter), and a Device Exclusion Vector (DEV). The DEV is an array of bits, one for each page of physical memory, that controls DMA to physical memory pages by DMA-capable devices. Physical memory is modeled as an array of one bit pages. One bit per page of physical memory is sufficient to model SecVisor's code integrity property. In our model, a physical page with value zero corresponds to an integrity-preserved page while a page with value one corresponds to an integrity violation.

```
-- Device Exclusion Vector
dev_t: array [ page_index ] of bit;
phy_mem_t: array [ page_index ] of bit;
-- Hardware Platform
hardware_platform_t: record
 phy_mem : phy_mem_t;
 mode: bit;   -- CPU Mode
 IP: word;    -- Instruction Pointer
 DEV : dev_t; -- Device Exclusion Vector
end;
```

Virtual Memory. We model virtual memory at the granularity of a page. Each page table entry includes a read/write bit, execute bit, and a physical address mapping.

```
pte_t: record -- Page Table Entry
 rw: bit; -- Read/Write
 x: bit;  -- Execute
 PA: word; -- Physical Address
end;
```

Page tables (PTs) are a basic data type and are modeled as arrays of page table entries.

```
page_table_t: array[index_PTEs] of pte_t;
```

Variables in our model include a hardware platform, actual world shadow page tables (SPTs) and kernel page tables (KPTs), and an ideal world copy of the shadow page tables called the secure SPT (denoted as `sec_SPT` in the code).

```
hw : hardware_platform_t;
-- Actual World
SPT : page_table_t;  -- Shadow PT
KPT : page_table_t; -- Kernel PT
-- Ideal World
sec_SPT : page_table_t; -- Shadow PT
```

3.3.2 Modeling SecVisor

We model SecVisor's initialization, kernel entry and exit handlers, and shadow page table synchronization code. For each component of the model, we ensure that it is a conservative abstraction of the corresponding fragment of SecVisor via manual inspection.

SecVisor Initialization. SecVisor's initialization code initializes SecVisor's shadow page tables under the assumption that the kernel executes first on system start up. This is captured in our model as follows:

```
procedure secvisor_init_kernel_mode();
begin
-- Start in kernel mode
 hw.mode := KRNL_MODE;
 hw.IP := KRNL_CODE;
-- Init. DEV to protect kernel code
 hw.DEV[KRNL_CODE] := 1;
-- 1) Initialize actual model PTs
-- User memory permissions RW
 SPT[USER_MEM].rw := RW;
 SPT[USER_MEM].x := NON_eXe;
 SPT[USER_MEM].PA := USER_MEM;
-- Kernel code permissions RX
 SPT[KRNL_CODE].rw := R;
 SPT[KRNL_CODE].x := eXe;
 SPT[KRNL_CODE].PA := KRNL_CODE;
-- Kernel data permissions RW
```

```
 SPT[KRNL_DATA].rw := RW;
 SPT[KRNL_DATA].x := NON_eXe;
 SPT[KRNL_DATA].PA := KRNL_DATA;
-- 2) Initialize ideal model SPT
-- Set user memory permissions to RW
 sec_SPT[USER_MEM].rw := RW;
 sec_SPT[USER_MEM].x := NON_eXe;
 sec_SPT[USER_MEM].PA := USER_MEM;
-- Set kernel code permissions to RX
 sec_SPT[KRNL_CODE].rw := R;
 sec_SPT[KRNL_CODE].x := eXe;
 sec_SPT[KRNL_CODE].PA := KRNL_CODE;
-- Set kernel data permissions to RW
 sec_SPT[KRNL_DATA].rw := RW;
 sec_SPT[KRNL_DATA].x := NON_eXe;
 sec_SPT[KRNL_DATA].PA := KRNL_DATA;
end;
```

The permissions set in the actual as well as secure shadow page tables correspond to those in Figure 1 for kernel mode execution.

Kernel to User Mode. On a transition to user mode, SecVisor's kernel exit handler sets: (a) the mode to user mode, and (b) the shadow page tables such that user memory becomes executable and kernel code pages become non-executable. This is modeled by the following Murφ procedure:

```
procedure secvisor_kernel_exit();
begin
-- User mode
 hw.mode := USER_MODE;
 hw.IP := USER_MEM;
-- Set user mem. to executable
 SPT[USER_MEM].rw := RW;
 SPT[USER_MEM].x := eXe;
-- Set kernel code non-exec.
 SPT[KRNL_CODE].x := NON_eXe;
 SPT[KRNL_DATA].x := eXe;
 SPT[KRNL_DATA].rw := RW
-- Set ideal world SPT
 sec_SPT[USER_MEM].rw := RW;
 sec_SPT[USER_MEM].x := eXe;
 sec_SPT[KRNL_CODE].x := NON_eXe;
 sec_SPT[KRNL_DATA].x := eXe;
 sec_SPT[KRNL_DATA].rw := RW
end;
```

Note that the Murφ code above only models the set of permissions that are required to achieve the properties of code integrity and execution integrity. The permissions in the SPT in the implementation of SecVisor are likely to be more restrictive. For example, a page containing kernel data is unlikely to be writable or executable in user mode. Modeling only the minimal set of permissions simplifies our formal model, which reduces the state space that Murφ needs to explore.

User to Kernel Mode. On a transition to kernel mode, SecVisor's entry handler sets the shadow page tables such that the kernel code becomes executable and the kernel data and user memory become non-executable. This prevents unapproved code from being executed during kernel mode, and is captured in our model as follows:

```
procedure secvisor_kernel_entry();
begin
-- Kernel mode
 hw.mode := KRNL_MODE;
 hw.IP := KRNL_CODE;
-- 1) Set actual model PTs
-- User memory permissions RW
 SPT[USER_MEM].rw := RW;
 SPT[USER_MEM].x := NON_eXe;
-- Kernel code permissions RX
 SPT[KRNL_CODE].rw := R;
 SPT[KRNL_CODE].x := eXe;
-- Kernel data permissions RW
 SPT[KRNL_DATA].rw := RW;
 SPT[KRNL_DATA].x := NON_eXe;
-- 2) Set ideal model SPT
-- User memory permissions RW
 sec_SPT[USER_MEM].rw := RW;
 sec_SPT[USER_MEM].x := NON_eXe;
-- Kernel code permissions RX
 sec_SPT[KRNL_CODE].rw := R;
 sec_SPT[KRNL_CODE].x := eXe;
-- Kernel data permissions RW
 sec_SPT[KRNL_DATA].rw := RW;
 sec_SPT[KRNL_DATA].x := NON_eXe;
end;
```

An interesting point about the code snippet above is that it explicitly sets the `hw.IP` variable to approved kernel code. In other words, the code assumes that property P1 is axiomatically satisfied. This is consistent with our modeling choices.

Page Table Synchronization. For correct operation, SecVisor synchronizes the shadow page table with the kernel pages table when the kernel: (i) wants to use a new page table, (ii) modifies an existing page table entry, and (iii) creates a new entry in its page tables. An attacker can modify the kernel page table entries. Hence, SecVisor must prevent the attacker's modifications from affecting the SPT fields that enforce code and execution integrity. SecVisor's design specifies that to prevent adversary modification of sensitive SPT state, SecVisor may not copy permission bits from the kernel page table during synchronization with the shadow page table. We model this in Murφ as follows:

```
procedure secvisor_spt_synchronize();
begin
 SPT[KRNL_CODE].PA := KPT[KRNL_CODE].PA;
 SPT[KRNL_DATA].PA := KPT[KRNL_DATA].PA;
 SPT[USER_MEM].PA := KPT[USER_MEM].PA;
end;
```

3.3.3 Modeling System Transitions

Our model includes two system transition rules that correspond to kernel entries and exits. These two rules enable the non-deterministic exploration of the entire reachable state space by Murφ. A transition rule includes a guard that must be satisfied for the rule to be enabled, and a command that yields successor states. In Murφ syntax, the guard and the command are separated by `==>`.

Kernel Exit. The kernel exit transition rule specifies that if the system is in kernel mode then a valid transition is to transition to user mode. The transition rule sets the IP to point to user memory while the system is executing in kernel mode. Recall that since SecVisor removes execute permission for user memory, the exit causes the CPU to throw an exception. SecVisor intercepts the exception an executes an exception handler, which we model with the `setIP(new_IP)` procedure. We model the entire kernel exit in Murφ as follows (note that the procedure `secvisor_kernel_exit()` is defined earlier in this section):

```
rule "Kernel Exit"
 hw.mode = KRNL_MODE ==> setIP(USER_MEM);
end;

-- Set IP to USER_MEM (unapproved)
procedure setIP(new_IP : page_index);
begin
 if (SPT[new_IP].x = NON_eXe) then
    secvisor_kernel_exit();
 else
    hw.IP := new_IP;
 endif
end;
```

Kernel Entry. The kernel entry transition rule specifies that if the system is in user mode then a valid transition is to kernel mode. The transition is modeled in Murφ by the procedure defined previously in this section:`secvisor_kernel_entry()`.

```
rule "Kernel Entry"
 hw.mode = USER_MODE ==>
   secvisor_kernel_entry();
end;
```

3.3.4 Modeling Adversary Actions

The adversary can write to memory pages with the read/write bit set, perform DMA writes to physical pages whose DEV bit is not set, arbitrarily modify kernel page-table entries, and create new kernel page table entries. We model the adversary's write abilities as follows:

```
rule "Attacker writes through MMU"
true ==> write();
end;

procedure write();
begin
for i : index_PTEs do
 -- If SPT entry is writable
 if (SPT[i].rw = RW) then
   -- Modify page integrity bit
   hw.phy_mem[SPT[i].PA] := 1;
 endif
endfor
end;

rule "Attacker attempts DMA write"
true ==> dma_write();
end;

procedure dma_write();
begin
for i : page_index do
 -- If DEV protections are not set
 if (hw.DEV[i] = 0) then
   -- Modify page integrity bit
   hw.phy_mem[i] := 1;
 endif
endfor
end;
```

We provide the adversary with the ability to arbitrarily modify every field of every kernel page table entry including the read/write permissions, execute permissions, and physical address mapping for user memory, kernel code, and kernel data. After modification, SecVisor synchronizes the kernel page table with the shadow page table as specified in SecVisor's design.

```
--General KPT attacker
ruleset va : index_PTEs  do
ruleset pa : page_index do
ruleset rw: bit do
ruleset x: bit do
  rule "Modify KPT entries"
   true   ==>
    begin
       -- Create arbitrary entry
      KPT[va].rw := rw;
      KPT[va].x := x;
      KPT[va].PA := pa;

      secvisor_synchronize();
    end;
endruleset; endruleset;
endruleset; endruleset;
```

3.3.5 Specifying Security Properties

We model SecVisor's security properties as invariants. Recall that an invariant is a condition that holds in all (and only) safe states of the model. Our invariants state a necessary relationship for security by comparing between the ideal world (without the adversary) and the actual world (with the adversary). Intuitively, our invariants stipulate that the actual world never deviates from the ideal world with respect to our desired security property, and thus is secure.

Execution Integrity. The execution integrity invariant $ExecInt$ states that if the system is in kernel mode then the IP must point to an approved code region. Thus, $ExecInt$ corresponds to SecVisor's P2 property. In our model, we express $ExecInt$ as an equality between the approved status of a page in the secure SPT and the approved status of the same page in the SPT of the actual model. The violation of $ExecInt$ implies the adversary has managed to alter the approved status of a page in a way that is disallowed in the ideal world, and indicates a possible attack on SecVisor.

```
hw.mode = KRNL_MODE ->
  sec_SPT[sec_SPT[hw.IP].PA].x
    = SPT[SPT[hw.IP].PA].x;
```

Code Integrity. The code integrity invariant $CodeInt$ states that approved code must not be modified by any agent other than SecVisor and its TCB. In our model, we initialize physical memory to all zeros. The attacker then attempts to set a kernel code page to one, and thus violate code integrity. The following invariant, which we use as $CodeInt$, states that SecVisor prevents the adversary from ever modifying approved kernel code, and hence approved code is always set to zero.

```
hw.phy_mem[KRNL_CODE] = 0;
```

3.4 Modeling Initial States

Our model begins with a call to `skinit()`, which in turn invokes `secvisor_init_kernel_mode()` as defined earlier.

```
startstate
begin
-- Model Hardware Reset
 clear hw; clear SPT;
 clear KPT; clear sec_SPT;
-- Start SecVisor in isolation
 skinit();
end;
```

Subsequently a transition rule fires and moves the system to a new state. Since the system is currently in kernel mode, the only valid transition rules are a kernel exit or one of the adversary's transition rules. If the kernel exit transition fires, the instruction pointer is set to point to user memory. If the page of user memory is marked as non-executable in the shadow page tables, then the SecVisor kernel exit handler is invoked.

Subsequently, the only valid transition is to re-enter kernel model which results in an execution of the SecVisor kernel entry handler. The entry handler sets the shadow page tables according to the code above, after which a non-deterministic transition occurs. The process completes when either an invariant is violated or when executing enabled transitions does not lead to any previously unexplored states.

4 Analysis

We used Murφ to check whether our SecVisor model satisfied the two security properties. Murφ identified an execution that violates $ExecInt$, and one execution that violates $CodeInt$. Based on these counterexamples, we identified *vulnerabilities* in SecVisor's design and implementation. Both vulnerabilities result from flaws in SecVisor's shadow page table synchronization code. Furthermore, we demonstrate that the vulnerabilities could be utilized by an attacker to take control of a SecVisor-protected kernel by implementing *exploits* and launching two successful code injection attacks against a SecVisor-protected Linux kernel. The rest of this section presents these vulnerabilities and exploits in detail.

4.1 Approved Page Remapping

Our first vulnerability, called Approved Page Remapping, was derived from Murφ's counterexample to $ExecInt$. The counterexample ends with the adversary modifying an approved kernel page table (KPT) entry. Specifically, the adversary alters the KPT entry that translates the virtual addresses of approved code to point to a physical page containing unapproved code. The modified KPT entry is then copied by SecVisor's synchronization procedure into the shadow page table (SPT). The key flaw here is that the KPT entry is copied into the SPT without checking that approved code virtual pages are not mapped to physical pages containing unapproved code. Subsequently, the CPU is in a position to execute unapproved (and possibly malicious) code. Figure 4 illustrates this attack.

Exploiting the Remapping Vulnerability. Our exploit for the Remapping flaw sets the physical address of a kernel page table entry for kernel code to point to user memory[4]. After synchronizing the SPT with the KPT, the re-

[4] Alternatively, the exploit could set the kernel code entry to point to kernel data.

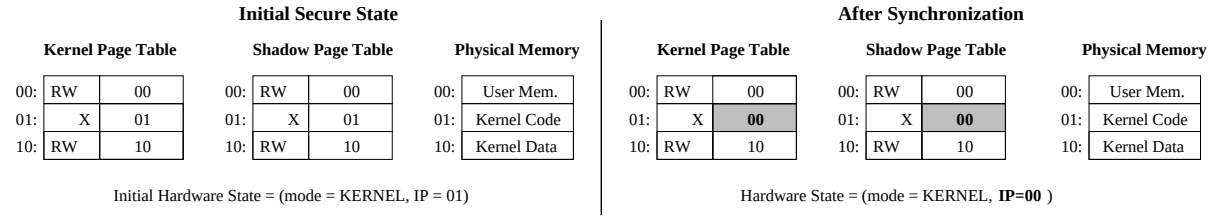

Figure 4: This figure depicts a transition from an initial secure state to the compromised state resulting from an Approved Page Remapping exploit. The attacker modifies the kernel page table entry 01 to point to the writable physical page 00. Subsequently, the CPU executes the unapproved code in page 00 in kernel mode.

sulting SPT entry for kernel code points to user memory, which is unapproved.

```
procedure remapping_exploit();
begin
-- Approved page --> user memory
 KPT[KERNEL_CODE].PA := USER_MEM;
-- Variant: Approved page --> kernel data
-- KPT[KERNEL_CODE].PA := KERNEL_DATA;
-- Synchronize SPT with KPT
 secvisor_synchronize();
end;
```

After adding the exploit to the adversary in the model, Murφ discovers a counterexample that identifies a specific attack which violates $ExecInt$. To further ensure that the Remapping Vulnerability is also inherent in SecVisor, we crafted a SecVisor exploit (about 37 lines of C) as an kernel module. This exploit modifies the physical address of a page table entry mapping an approved code page, to point to a page containing unapproved code. When executed on a SecVisor-protected Linux kernel running on an AMD SVM platform, our exploit overwrote a page table entry which originally mapped a physical page containing approved code to point to an arbitrary (unapproved) physical page. SecVisor copied this entry into the SPT, potentially permitting the CPU to execute unapproved code in kernel mode.

Our current exploit implementation requires that SecVisor approve the kernel module containing the exploit code for execution in kernel mode. This is unrealistic since any reasonable approval policy will prohibit the execution of kernel modules obtained from untrusted sources. However, it could be possible for the attacker to run our exploit without loading a kernel module. This could be done by executing pre-existing code in the kernel that modifies the kernel page table entries with attacker-specified parameters. The attacker could, for example, exploit a control-flow vulnerability (such as a buffer overflow) in the kernel to call the kernel page table modification routine with attacker-supplied parameters.

4.2 Writable Virtual Alias

Our second vulnerability, called Writable Virtual Alias, was derived from Murφ's counterexample to $CodeInt$. The counterexample indicates that the adversary can create a writable virtual alias to (i.e., another page table entry with write permissions pointing to) a physical page containing approved code. SecVisor's synchronization code copies the alias into the shadow page table without checking that the alias is pointing to an approved physical page. Then, the adversary uses the writable virtual alias to inject arbitrary code into approved kernel code pages. The result is a violation of code integrity. Figure 5 illustrates this attack.

Exploiting the Alias Vulnerability. Our exploit causes a writable virtual page of user memory to point to kernel code. After synchronization, the resulting SPT entry for user memory points to an approved physical page of kernel code.

```
procedure alias_exploit();
begin
-- Create writable alias to approved page
 KPT[USER_MEM].rw := RW;
 KPT[USER_MEM].x := NON_eXe;
 KPT[USER_MEM].PA := KERNEL_CODE;
-- Synchronize SPT with KPT
 secvisor_synchronize();
end;
```

After adding the exploit to the adversary in the model, Murφ discovers a counterexample that identifies a specific attack which violates $CodeInt$. To further verify that this flaw is also inherent in SecVisor, we created an exploit using about 15 lines of C code. Our exploit opens `/dev/mem`, maps a user page with write permissions to a physical page (at address `KERNEL_CODE_ADDR`) containing approved kernel code, and writes an arbitrary value into the target physical page via the virtual user page. When executed against SecVisor on an AMD SVM platform running Linux 2.6.20.14, our exploit successfully overwrote an approved kernel code with arbitrary code. An interesting aspect of the exploit is that it can be executed from user mode by an attacker that has administrative privileges.

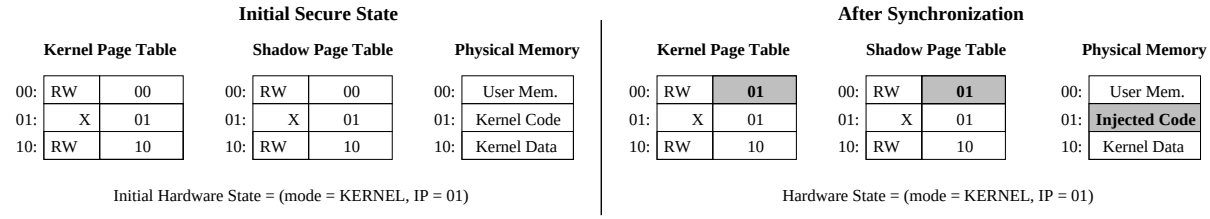

Figure 5: This figure depicts a transition from an initial secure state to the compromised state resulting from a Writable Virtual Alias exploit. The attacker modifies the kernel page table entry 00 to be a writable alias for physical page 01 then injects code into 01 using the alias.

```
void remap_kernel_code() {
  // Open /dev/mem device
  fd = open("/dev/mem", O_RDWR);
  // Map kernel code with RW access
  // into user address space
  user_mem = mmap(0,
         KERNEL_CODE_SIZE,
         PROT_READ|PROT_WRITE,
         MAP_SHARED,fd,
         KERNEL_CODE_ADDR);
  // Overwrite kernel code
  memset(user_mem, ATTACK_CODE_ADDR,
         ATTACK_CODE_SIZE);
}
```

5 Repairing and Verifying SecVisor

In this section, we describe the design and implementation of a secure synchronization procedure that repairs both the vulnerabilities described in Section 4. We fix SecVisor's implementation by modifying 105 lines of C code. We verify the correctness of our fix by augmenting the formal model; Murφ no longer finds any attacks. We further check that our exploits on the original version of SecVisor no longer compromise code or execution integrity in the repaired version. We demonstrate that our defense is efficient by running benchmarks on the repaired implementation of SecVisor; the performance loss due to our fix ranges from 1–7%.

5.1 Design of Defense

The vulnerabilities in SecVisor are due to the fact that the SPT synchronization procedure copies untrusted data into the SPT without validating the data fully. A secure synchronization must validate all fields in each entry of the kernel page tables before copying that entry into the SPT. In the original version of SecVisor, the SPT synchronization procedure misses two checks that lead to the two vulnerabilities we describe in Section 4. Therefore, our fix introduces the two checks into SecVisor's SPT synchronization procedure. Since Murφ is no longer able to find any attacks that violate either code or execution integrity, we obtain the guarantee that the SPT synchronization procedure is correctly validating the kernel's page table entries. The two checks our fix introduces are: 1) all virtual aliases of an approved physical page must disallow writes to the physical page, and 2) kernel page table entries that map approved virtual pages must contain addresses of approved physical pages.

The pseudo-code for these two checks appear below. SecVisor internally maintains the list of mappings between approved virtual pages and approved physical pages. This list is referred to as the `refset` in the code given below. Further, `refset.virt` refers to set of all virtual addresses in the `refset` and `refset.phys` refers to all physical addresses in the `refset`. PA refers to the physical address field of a page table entry. The function `virt` takes an index into a page table as input and returns the virtual address corresponding to that index.

```
procedure check_aliasing()
begin
  for i in index_PTEs do
    //alias of approved physical page?
    if (KPT[i].PA ∈ refset.phys)
      //remove write permission
      SPT[i].rw := R;
      SPT[i].x := eXe;
      SPT[i].PA := KPT[i].PA;
      // add new mapping to refset
      refset ← refset ∪ {virt(i),KPT[i].PA}
    endif
  endfor
end;

procedure check_phy_addr()
begin
  for i in index_PTEs do
    //mapping of approved virtual page?
    if (virt(i) ∈ refset.virt)
      SPT[i].PA := refset[virt(i)].phys
    endif
  endfor
end;
```

Verification	States	Rules Fired	Time	Memory	Result
3 PTEs	55,296	2,156,544	2.52 sec.	8MB	Success
4 PTEs	1,744,896	88,989,696	343.97 sec	256MB	Success

Figure 6: Verification results for models with 3 and 4 page table entries including the number of states searched, number of rules fired, the time required, and the memory required.

We added the two checks shown above to SecVisor's SPT synchronization procedure by modifying 105 lines of C code. We verified that our exploits failed against the patched version of SecVisor.

5.2 Verification

We augment our model with the defense and perform a number of verifications with Murφ. We ran each verification on a 3.20GHz Pentium 4 with 2GB of memory. Figure 6 details our results. Each verification includes three pages of physical memory representing kernel code, kernel data, and user memory. In the first verification, we include three page table entries in both the kernel page tables and the shadow page tables. Murφ reported a successful verification after searching over 55,000 states, firing over 2 million rules, and running for 2.5 seconds with a maximum memory utilization of less than 8MB. We increased the number of page table entries in both the kernel page table and the shadow page table to four entries and reran the verification. Since the adversary can arbitrary modify every bit of a kernel page table entry (about 4 bits) and we add two additional pages, we expect the resulting model to be at least 9 times larger than the previous model, but no more than 2^7 times larger. Murφ successfully verified the four page model after searching over 1.7 million states, firing more than 88 million rules and completing in around 6 minutes with a maximum memory utilization of less than 256MB. Compared to the three entry verification, the resulting verification searched approximately 2^5 times more states and required 2^5 times more memory. We attempted to add a fifth page table entry, but the verification exceeded the memory capacity of the machine. Given the exponential dependence on the number of page table entries, a five page verification would require close to 8GB of memory. Continuing to verify larger models adds little insight into SecVisor's security. Since the adversary's abilities do not fundamentally increase with additional pages, the same attacks exist regardless of the number of page table entries.

5.3 Performance Evaluation

In order to evaluate the performance impact of our security fixes, we ran kernel microbenchmarks, standardized benchmarks, and application benchmarks. We use the same benchmarks as were used in the original SecVisor paper. We find that the overhead of our security fixes is less than 7%. Our experimental platform is the HP Compaq dc5750 Microtower PC. This PC uses an AMD Athlon64 X2 dual-core CPU running at 2200 MHz with 2 GB of RAM and runs the i386 version of the Fedora Core 6 Linux distribution. All our experiments run on the uniprocessor version of Linux kernel 2.6.20.14, which executes on top of SecVisor. The source of the overhead in the patched version of SecVisor are the extra checks introduced into the SPT synchronization procedure. Therefore, it is natural that workloads with rapidly varying working sets, which increase the rate of updates to the kernel's page tables, will exhibit the maximum overhead.

lmbench Microbenchmarks. We use the lmbench benchmarking suite to measure overheads of different kernel operations when using SecVisor with vulnerabilities (SecVisor-vul) and the patched version SecVisor (SecVisor-fix). The worst case overhead is about 7%. Our results are presented in Figure 7.

Application Benchmarks. We execute both compute-bound and I/O-bound applications in order to evaluate the overhead of our security fixes. For our compute-bound applications we choose benchmarks from SPECint 2006 suite. Our I/O bound applications consist of the gcc benchmark from SPECint 2006, Linux kernel compile, unzipping and untarring the Linux kernel source, and the Postmark file system benchmark.

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 20,000 files, 100,000 transactions, and 100 subdirectories, and all other parameters are set at their default values. We run each of these applications five times each on the vulnerable version of SecVisor and the patched version of SecVisor.

Our results are presented in Figure 8. We show the result of only one of the compute-bound SPEC benchmarks since none of these benchmarks exhibit any overhead. The gcc SPEC benchmark, Linux kernel compile, and Postmark which are I/O bound have greater overheads than the compute bound SPEC benchmarks. This is because of rapidly varying working sets which leads to frequent SPT synchronization operations.

Null Call	Fork	Exec	Prot Fault	PF	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K
1.0211	1.0170	1.0250	1.0217	1.0652	1.0690	1.0551	1.0662	1.0613	1.0687

Figure 7: Normalized execution times of `lmbench` process, memory and context switch microbenchmarks. All times are normalized with respect to the execution times of the vulnerable version of SecVisor. PF stands for page fault.

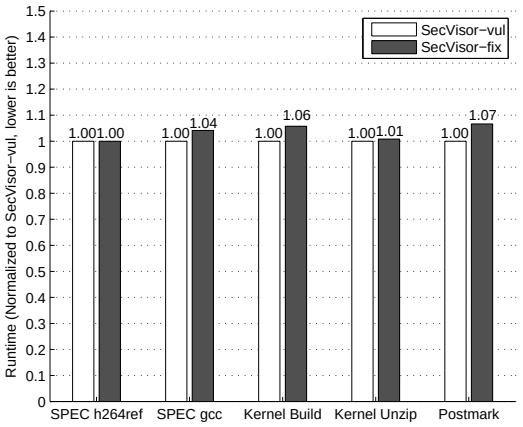

Figure 8: Application performance comparison between SecVisor-vul and SecVisor-fix, normalized to SecVisor-vul.

6 Lessons Learned

Given the choice between manual audit and formal verification, secure system designers overwhelmingly choose manual audits, despite well-known weaknesses [15], for several specific reasons. While formal analysis can result in higher-assurance systems, it requires considerably more investment in terms of both time and technical effort, making it less practical. Formal analysis requires not only an understanding of the system design, but also that the design be translated to a formal model in a way that preserves the desirable security properties of the original system.

During this project, we identified the following three general facets of secure system design that reduce the code size and complexity of verification, while facilitating an accurate correspondence between our model and the design: (1) clear specification of the adversary model, (2) explicit statement of the top-level security requirements that the system aims to provide, and (3) clean separation between these requirements and the mechanisms used to achieve them. We now elaborate on these facets since we believe that they are relevant to the design, modeling and verification of secure systems in general.

Explicit Adversary Model. In SecVisor's design, the adversary is endowed with two distinct abilities: using DMA to write to memory and modifying the kernel's page tables. These abilities correspond exactly to the external interface SecVisor provides. By using a narrow interface between trusted and untrusted domains, the capabilities of the adversary are clearly specified. If the adversary's interface or equivalently, the interface between trusted and untrusted domains, is underspecified, the implementation of the adversary model is both more complicated and less likely to accurately reflect reality.

Top-Level Security Requirements. The security of SecVisor depends on two requirements, code integrity and execution integrity. SecVisor's specification clearly relates these requirements with its security goals. The explicit statement of requirements helped focus our verification. In our model, these requirements lead to properties, which in turn map to invariants with minimal modifications. Such a clean mapping inspires confidence that a successful verification implies a secure system design.

Separation of Requirements and Mechanism. One aspect of SecVisor's design that was clarified during our analysis was the interplay between design requirements and the mechanisms used to achieve them. In the original presentation of SecVisor's design [17], the two top level requirements are decomposed to properties **P1-P4**. During our verification, we see that properties **P1-P3** are actually mechanisms used to achieve execution integrity, while **P4** is the mechanism to achieve code integrity.

7 Related Work

Our use of two models – ideal and actual – is inspired by the work of Lie et al. on specifying and verifying XOM [11] using Murφ. XOM is a hardware-based approach to ensuring tampering-resistance and copy-resistance. In contrast, SecVisor aims at achieving memory protection. Murφ has also been used by Mitchell et al. [12, 13] to successfully verify the correctness of (and find bugs in) security protocol specifications. Even though we use Murφ, our approach is amenable to verification via other model checkers, such as SPIN [8], TLA+ [10] and SMV [5].

A number of projects [6, 2, 21] have used software model checking and static analysis to find a general class of bugs in source code, without a specific attacker model. In contrast, our focus is on bug detection, and verification, in the presence of an adversary with precisely defined capabilities.

[5] `http://www.cs.cmu.edu/ modelcheck/smv.html`

Prior work has explored the problem of verifying the design of secure systems [14, 16, 18]. These works are similar to our own in spirit, but differ in the methods applied. They suggest an approach where properties are manually proven using a logic and without an explicit adversary model. In contrast, our focus is on automated verification of manually constructed models that include an explicit adversary model.

Heitmeyer et al. [7] used the PVS theorem prover to verify the correctness of a data separation kernel. In contrast, we use model checking to verify the correctness of a hypervisor aimed at memory protection.

8 Conclusion

We verified the design of SecVisor, a security hypervisor, using the Murφ model-checker. During the verification, Murφ identified executions that violated both of the desired security requirements of SecVisor—execution integrity and code integrity. Based on these design-level attacks, we crafted 2 exploits and successfully launched code injection attacks on a SecVisor-protected Linux kernel. The exploits consisted of 37 and 15 lines of C code respectively. We repaired SecVisor by designing and implementing a secure memory protection scheme by modifying 105 lines of code. We also incorporated the fix in the formal model and verified using Murφ that the resulting design is secure. The verification completes in about 6 minutes after exploring approximately 1.7 million states. The repaired implementation is still efficient: performance loss is in the range of 1-7% on a standard benchmark suite for SecVisor. All source code for this project is available at `www.cs.cmu.edu/~jfrankli/secvisor`. Finally, we identified general facets of secure system design that facilitate formal analysis.

References

[1] ADVANCED MICRO DEVICES. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.12 ed., September 2006.

[2] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security* (2002), pp. 235–244.

[3] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2008), ACM, pp. 2–13.

[4] DILL, D. L. The mur*phi* verification system. In *CAV* (1996), pp. 390–393.

[5] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 17–30.

[6] HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)* (Berlin, Germany, June 2002), vol. 37(5) of *SIGPLAN Notices*, Association for Computing Machinery, pp. 69–82.

[7] HEITMEYER, C. L., ARCHER, M., LEONARD, E. I., AND MCLEAN, J. D. Formal specification and verification of data separation in a separation kernel for an embedded system. In *ACM Conference on Computer and Communications Security* (2006), pp. 346–355.

[8] HOLZMANN, G. J. The Model Checker SPIN. *IEEE Trans. Software Eng. 23*, 5 (1997), 279–295.

[9] INTEL CORPORATION. Trusted eXecution Technology – Preliminary Architecture Specification and Enabling Considerations. Document number 31516803, Nov. 2006.

[10] LAMPORT, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] LIE, D., MITCHELL, J., THEKKATH, C. A., AND HOROWITZ, M. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (2003).

[12] MITCHELL, J. C., MITCHELL, M., AND STERN, U. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997).

[13] MITCHELL, J. C., SHMATIKOV, V., AND STERN, U. Finite-State Analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium* (1998), pp. 201–216.

[14] NEUMANN, P., BOYER, R., FEIERTAG, R., LEVITT, K., AND ROBINSON, L. A provably secure operating system: The system, its applications, and proofs. Tech. rep., SRI International, 1980.

[15] OPENBSD.ORG. Source code auditing process. http://www.openbsd.org/security.html#process, May 2008.

[16] RUSHBY, J. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles (SOSP)* (Asilomar, CA, Dec. 1981), pp. 12–21. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[17] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

[18] SHAPIRO, J. S., AND WEBER, S. Verifying the eros confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (2000).

[19] WITCHEL, E., RHEE, J., AND ASANOVIĆ, K. Mondrix: memory isolation for linux using mondriaan memory protection. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005), 31–44.

[20] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), ACM, pp. 71–80.

[21] YANG, J., TWOHEY, P., ENGLER, D. R., AND MUSUVATHI, M. Using Model Checking to Find Serious File System Errors. In *OSDI* (2004), pp. 273–288.

[22] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI* (2006), USENIX Association, pp. 263–278.