

A Lightweight Hypervisor for Malware Analysis

Apeksha Godiyal, Anh Nguyen, Nabil Schear
Department of Computer Science
University of Illinois at Urbana-Champaign
{godiyal2, anguyen7, nschea2}@uiuc.edu

Abstract

Malicious software is rampant on the Internet and is costing billions of dollars each year. Safe and thorough analysis of malware is key to protecting systems and cleaning those that have already been infected. Virtualization offers strong isolation to malware analyzers, but most analysis tools must still run alongside the malware increasing detectability of the analysis platform. This reduces the value of the analysis because some malware is known to behave differently when being analyzed. Because of their complexity and generality of purpose, current general purpose virtualization platforms are not well suited to supporting customized analysis tools at the virtual machine monitor (VMM) level.

We propose a lightweight hardware-supported virtualization platform that is purpose-built for malware analysis. Hardware virtualization makes the VMM difficult to detect and reduces its size and complexity. We further simplify our VMM by not implementing virtualization features that are unnecessary for malware analysis (e.g., virtual device emulation). Our platform is more amenable to developing and deploying analysis techniques directly in the VMM than Xen or VMWare. In this paper, we discuss our prototype design and implementation. We also discuss the effectiveness of various malware analysis techniques that we have developed to run on our platform. Finally, we show that our platform is not susceptible to general purpose virtualization detection techniques.

1 Introduction

In today's hostile but highly connected computing environment, spyware, adware, viruses, and trojans – collectively called malware, pose a serious threat to our computer systems. According to Computer Economics, more than 13 billion dollars were lost due to malware attacks in 2006 alone [7]. Furthermore, 55% of all on-line users be-

lieve their systems had been infected with spyware [21]. All signs point towards malware being a considerable risk in the future.

Malware analysis plays a critical role in countering this alarming trend. Through detailed analysis of a particular malicious application, security researchers are able to gain an insightful view of its intent, mechanisms, and risk. This knowledge is very valuable in predicting the threats posed by the malware, creating anti-virus signatures, developing tools to patch infected systems, and in some cases tracing back to the criminal behind it. Traditional tools for malware analysis include disassemblers [26], debuggers [34], dynamic black box analysis such as function call tracing facilities (e.g., strace), and network sniffers [33]. While these methods have proved to be somewhat effective, each suffers from certain drawbacks. Disassembling, like other static analysis techniques, can be circumvented by packing or dynamic code translation [17, 23]. Dynamic black box analysis gives only an incomplete view of the malware. Debugging, on the other hand, provides a more exhaustive view but is vulnerable to debugger fingerprinting [29, 12]. As malware gets more and more complex, it is often impractical and unnecessary to analyze each and every instruction. The best general solution is to combine these techniques.

Another common approach in malware analysis is to use analyzing tools in conjunction with virtualization technology, taking advantage of its capabilities to isolate and roll back the system hosting the malware. Because of the simple resources abstraction provided by virtual machines, many systems use them to perform intrusion detection [11, 6, 20, 14], while others use them to create honeypots [30, 32]. Unfortunately, malware writers are using increasingly complex techniques to evade detection and prevent forensic analysis. To avoid being analyzed in a virtual machine, attackers have developed techniques to detect virtualization through side channels [24] or through artifacts of the virtualization platform [18]. When the malware detects it is running inside a

virtual machine, it will often exit and prevent further analysis of its activities. Some malware is even known to behave differently to fool analyzers of its intentions [8]. Therefore, minimal detectability of the virtualization platform will greatly affect the outcome and the accuracy of malware analysis. Existing virtual machine monitors (VMMs) such as VMWare or Xen, which are often used in malware analysis, are very vulnerable to detection mainly due to their generality of purpose. By taking advantage of recently introduced hardware-support for virtualization [13, 3] and focusing on malware analysis only, we are able to create simpler virtualization platform which will be both easier to use/extend and harder to detect.

In this paper, we present the design and implementation of a hardware virtualization supported malware analysis platform which we call MAVMM (Malware Analysis Virtual Machine Monitor). Our goals for this work are

1. small and simple hypervisor
2. minimal detectability
3. minimal dependence on the structure and trust of the guest operating system

Our platform uses a purpose-built hypervisor which allows a human analyzer access to the monitored guest's activities. Our system works by extracting various *features* from the monitored guest operating system. We have investigated how to support extraction of the following features: memory pages, disk blocks, system calls, function calls, and network interaction.

Using these low-level abstractions, we can provide the analyzer with a fairly complete picture of the malware's activities while not trusting the guest operating system. Hardware enforced isolation protects the hypervisor from attack by the malware. While complete undetectability is most likely a panacea [10], our system is more difficult to detect than simply running analysis tools alongside the malware or using a standard virtualization system. Our approach improves upon previous work because of its limited detectability and simplicity for deploying new analysis techniques. Since this system will not support arbitrary virtualization features, we avoid the complexity and overhead of such systems (e.g., virtual device emulation). Our platform can be more readily accessible for analysis tools to be deployed than a full virtualization platform. It is also simple enough for other researchers to re-use for their own analysis techniques.

The remainder of this paper is organized as follows: We present the design for our VMM in Section 2. We describe our implementation and some of our experiences

during the development process in Section 3. We perform some preliminary evaluations on MAVMM in Section 4. We further examine related work in Section 5 and conclude in Section 6.

2 Design

In developing the design for MAVMM, we evaluated multiple mechanisms and techniques for feature extraction, communication with the analysis platform, and virtualization support. In this section, we present these techniques and mechanisms, explain the decision that we had to make, and why we chose one approach over the others.

2.1 Hardware Virtualization Support

We implement our system using the AMD Secure Virtual Machine (SVM) extensions. Because the memory management unit (MMU) is on die, AMD engineers were able to include more advanced virtualization features than the comparable Intel extensions (which do not have an on-die MMU). For example, AMD natively supports nested paging in hardware. It also provides convenient mechanisms to reserve physical memory from being used for DMA. Though we believe our system could be ported to the Intel platform, we found that it is easier to develop for the AMD architecture. AMD also provides the Simnow simulation environment [1] which fully supports the SVM virtualization extensions easing our development and testing.

2.2 Boot Strapping

Our VMM platform is based on TVMM [16] and currently runs inside of the AMD Simnow simulator. However, our VMM does not rely directly on Simnow to function and should be able to run on bare hardware. Our platform is booted directly by the boot loader to prevent any malicious application from subverting our system. Furthermore, the guest operating system is fully virtualized from boot time to ensure that the VMM is protected by the hardware.

We use the GRUB boot loader to start the system. The guest operating system image is specified using the module parameter in the GRUB configuration. GRUB passes a multiboot info structure to the VMM which defines the memory map, command line arguments, and any additional GRUB parameters that we may specify. Our VMM executable is stored in a simplified 32-bit ELF format. We use the `mkelf32` utility included with Xen to build this simplified ELF image from raw object files.

The guest operating system must also be an ELF image. By default, the Linux kernel has its own file format

that is different from ELF. However because ELF compatible kernel images are needed for other systems like network boot and kernel-in-BIOS, we use the `mkelfImage` tool to create an ELF image from the original Linux kernel. It combines the kernel image with a small stub loader which handles command line arguments and embeds an `initrd` file inside the ELF image.

We use nested paging to protect VMM memory from being tampered by the guest operating system. Before booting the guest, we update BIOS provided E820 memory map to mark host's memory pages as reserved. This gives the guest an illusion that these pages are used by the BIOS and therefore should not be altered.

To protect the VMM from being affected by external device DMA, we use the Device Exclusion Vector (DEV) feature of the AMD SVM extension. We allocate a bitmap which defines which memory pages are available for external DMA. We mark all VMM pages as unavailable by setting our modified DEV to one of the DEVBASE registers using the DEVCTL PCI configuration space function block [2].

2.3 Memory

Memory is the most important feature that we need to extract to analyze the running malware; it is also critical for other types of analysis like call tracing and network interaction. Thankfully, AMD SVM provides an extensive interface for managing virtual machine memory. Our VMM uses nested paging rather than shadow paging. This adds an additional table and layer to the address translation process called the nested layer. The nested page table translates from guest physical to host physical addresses (see Figure 1). Hardware support helps to avoid excessive entries into the VMM while running memory intensive applications because the hardware can handle the translation directly and take advantage of the TLB cache.

In Linux (and most other modern operating systems), each process has its own virtual address space and page tables. We use the page table base pointer (stored in register CR3) to track a specific process. For now, we will assume that we already know the base pointer address for the malware process we wish to track (for further discussion on this see Section 2.7.2). To watch a process, we register a non-restartable intercept for the task switch event. We do not intercept on MOV to/from CR3 instructions because SVM does not execute that intercept on context switches and it would negate the performance improvement from nested paging by forcing a VMM entry on all translation events. We then examine the gCR3 register to get a pointer to the current processes page tables. If the process is being watched, we can dump the memory or otherwise examine its contents.

The most effective means of analyzing the memory of watched process in our system is to extract it from the analysis platform. This way we can use existing powerful tools on a normal machine to analyze the dump. We can dump process memory periodically or at specified times during the execution of a process. To begin the memory dump, we mark all the active mapped pages of the malware's virtual memory in the VMM host page tables as not writable. We also mark those pages as being copied using the OS reserved protection bits in the page table entry. We use this extra bit to allow the VMM to incrementally write the memory dump to persistent storage while the process continues to execute in copy-on-write manner. On each VMM entry (caused by intercepts, handling interceptable interrupts, etc..) the VMM incrementally dumps marked pages to persistent storage (see Section 2.7.1). If the VMM receives a page fault for a dumped page, the VMM will immediately write that page to persistent storage and clear the copy and write-only bits. To prevent large timing discrepancies, we copy a fixed number of memory pages to persistent storage per VMM entry. This allows us to amortize the cost of dumping the data across many VMM invocations (and even in other unwatched processes) which will appear like a much more plausible system slow down than an artificially long context switch time. If the persistent storage mechanism is still too slow to write, we can also copy the process memory to a scratch location inside the VMMs memory that is not available to the guest. This would allow multiple memory dumps to be pending writing to storage without slowing down the watched process considerably.

Since we are using the page table base address as the identifying feature of a watched process, we won't be able to monitor the memory of any sub-process which the malware creates. To combat this problem, we have considered two solutions: 1) dump all processes once the watched process begins and 2) infer sub-process CR3 addresses using system call tracing. While easy to implement, option 1 would produce higher performance overhead since persistent storage is likely to be slow. Furthermore, the voluminous data would be mostly unused and uninteresting while requiring considerable overhead. For these reasons, option 2 is more appealing because it allows us to reliably determine sub-process executions and track memory in a more efficient manner.

To accomplish sub-process memory tracking we need to intercept fork system calls (see Section 2.4). We keep a list of all *known* CR3 addresses for unwatched processes (including the kernel). When we intercept a fork system call we monitor all task switches during that processes execution for a new unknown address to be loaded into CR3. This way we can track sub-process executions that are the result of the malware application. Sim-

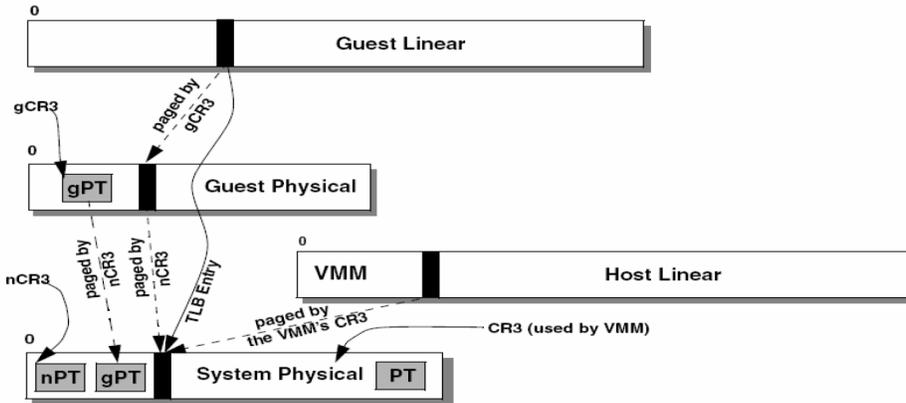


Figure 1: Nested Paging in AMD SVM [31]

ilar techniques have been demonstrated for tracking processes from a VMM in Antfarm [15].

2.4 System Calls

Native applications can invoke a system call in two different ways in Linux. By executing the `int $0x80` assembly language instruction, or by executing the `sysenter` instruction (`sysenter` is a recent addition and is only supported in 2.6 series Linux kernels). Similarly, the kernel can exit from a system call by executing the `iret` or the `sysexit` assembly language instructions [4].

Linux uses the `eax` register to pass the system call number from the user program to the kernel. The user mode process also finds the return code of the system call in the `eax` register. We have to access the system call number to identify the system call invoked. This is straightforward in case of the `int $0x80` instructions. The AMD SVM allows software interrupt instructions in the guest to be intercepted by means of control bits in the VMCB (Virtual Machine Control Block). `INT n` and `iret` instructions can be chosen by the hypervisor to be intercepted. In the VMCB control area, setting the 20th and the 21st bit of byte `0x00Ch`, enables `iret` and `INTn` instruction interception respectively [2].

In the event of an `INT` intercept, the hypervisor first checks if the instruction is `int $0x80`, by examining the 1st byte of the `EXITINTINFO` field in the VMCB (This byte is equal to the 8-bit `IDT` vector of the interrupt). If it is indeed `$0x80`, it reads `eax` register and can record the system call number. If the instruction is not `int $0x80`, the control is transferred to the guest. In the event of an `iret` intercept, the hypervisor records the `eax` value if the last intercept was an `int $0x80`. This state is maintained in the hypervisor in a single byte flag `int80`. After recording the return value, `int80` is reset.

Unfortunately, the hypervisor cannot register `sysenter/`

`sysexit` for interception. In cases where both the CPU and the Linux kernel can support `sysenter/sysexit` instruction, the libc wrapper function may generate them to invoke a system calls because they are faster than `int` and `iret`. We have not yet determined how to provide the system call analysis of `sysenter` and `sysexit`. We can either modify the Linux kernel to use only `int $0x80` or make it call a into the hypervisor just before the `sysenter` call and pass the exit code of `int $0x80` to the hypervisor (by writing the `EXITINTINFO` field of the `vmcb`). To avoid modifying the guest operating system heavily, we are currently planning to use Damn Small Linux (DSL) 4.2.5 which utilizes a 2.4 based kernel and `int $0x80` for system calls.

2.4.1 Parameter Gathering

The system call parameters are written in the CPU registers before issuing the system call. As with every C function call, parameters are automatically saved on the user stack when the wrapper routine (`_syscall0` etc) is invoked. This routine will find the parameters on the user stack, copy them to the appropriate registers and pass the parameters to the kernel appropriately.

Six registers are set aside to transfer the parameters. For system calls that require more than six parameters, a single register is used to point to a memory area in the process address space that contains the parameter values [4]. On a system call intercept, the hypervisor dumps the values in the six parameter registers. However, there is a semantic gap here. The hypervisor has no way of knowing the number and type of all the parameters. To overcome this, we can have a map in the hypervisor storing the number and type of each parameter, corresponding to all the system calls. This is similar in principle to the way the `strace` command works.

In case the last register points to a memory area containing more parameters, the hypervisor should retrieve

them directly out of the guest virtual memory. This is complicated by the fact that the page containing that memory area might get swapped to the disk. The chances of that page being swapped out is small as it was last accessed by the wrapper function, which then called `int $0x80`. In case that the page is invalid and swapped to disk, we mark it as needing a system call evaluation by the hypervisor (in the VMM page tables). We also mark the page as invalid in the VMM page tables so that once it is remapped by the guest operating system it will fault immediately to the VMM. When the guest kernel makes the page valid again, the system will fault into the hypervisor and dump the rest of the system call arguments.

2.5 Library Calls

We have investigated how to provide a list of library calls executed by the malware. However, there is a semantic gap to be bridged and we faced some design trade-offs. We studied how `ltrace` and `gtrace` utilities work [22]. `Ltrace` intercepts and displays all calls to dynamically linked functions, by using the PLT/GOT mechanism. `.plt` and `.got` are elf object headers that are used by the dynamic linker to resolve external/shared library references.

`Gtrace` traces all function calls and memory accesses of dynamically linked programs. We briefly describe how `gtrace` utility achieves function tracing [22]. It uses symbol information (to mark which symbol is a function) present in the `.symtab` and the `.dynsym` (for external and dynamic symbols exported). Then it inserts breakpoints in the entry point of all functions. The traced program is controlled with the `SYS_ptrace` system call from a different process. It also uses the `.plt` and `.got` sections to find address to insert breakpoints in dynamically linked shared function calls.

Let us reason how a similar method can be achieved in the hypervisor. Fortunately, the `.dynsym` and `.dynstr` sections are present in the running process image and the kernel can be modified to pass a pointer to the beginning of these sections to the hypervisor. The `.dynstr` section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries [28]. So the hypervisor can identify which library call is being invoked with the help of these two sections. But first the hypervisor needs to gain control every time a library call is made by the traced process. The function call instructions `JMP` and `CALL` cannot be registered for interception by the hypervisor. So we need to insert breakpoints in the entry point of all library calls, before the process starts executing. This has to be done by the kernel (by using `ptrace`) or through binary translation, which makes this method similar to debugging a process. We know that debug-

ging is vulnerable to debugger fingerprinting [12, 29]. So we have to make a trade off between what facilities we want to provide towards malware analysis and how light weight and undetectable the hypervisor should remain. We believe that system call tracing is more valuable for determining the effects of a malware application because they show all persistent or external interactions. While library call tracing is useful for reverse engineering how a malware application works, it is often unnecessary to assess its risk and impact.

2.6 Network and Disk

The support for tracking network interaction is based on tracking system calls. The Linux network API depends on the system call `sys_socketcall()` which in turn calls `sys_recv()` and `sys_send()`. We have discussed in section 2.4 how the hypervisor will track all system calls and gather their parameters. There are a few additional things the hypervisor should do to provide comprehensive network sniffing.

The hypervisor compares the `eax` register to the system call number for `sys_socketcall` i.e., 102. If the system call is `sys_socketcall`, it will do additional parsing of the parameters like obtaining the destination/source IP address by following the `sockaddr` pointer and the write message by following the write buffer pointer. This entails getting the data from the guest pages. However, that page might have become invalid due to swapping. This can be handled in the same manner as described in section 2.4.1. The hypervisor should store the pointer parameters for some calls like `recv()`, `recvmsg()` etc, which point to the received message buffers.

The `iret` case requires special treatment in case of `sys_socketcall`. The hypervisor will be in a state which tells that it is waiting for an `iret` interception. When an `iret` interception occurs, it follows the pointers stored from the parameters of `recv()`, `recvmsg()` calls and dump the buffers from guest memory. Unfortunately, the parsing of the return values and parameter structures will have to be specific to Linux network system calls.

2.7 VMM I/O

One of the questions that we faced in our design process is how to get data out of our system, so that future analysis could be done upon the data.

2.7.1 Getting Analysis Data out of VMM

We had several choices for extracting data from our analysis platform: use the same hard-disk as the guest operating system, a separated hard disk, a USB flash drive, or simply dumping the data out through the serial port. For

simplicity, we decided to avoid directly using the hard disk of the guest operating system to avoid conflicts and contention. Another alternative would be to modify the kernel and have it dump the data out. If the VMM contained its own hardware drivers and file systems, implementation would have been easier. However, this would go against our goal of making the platform as simple as possible. We also did not want to depend on the guest operating system to perform I/O for us because of trust issues and portability.

This leaves external drives and serial port communications as the preferred methods for extracting data. For both external drive and serial port, we can use BIOS services to dump the data out. With the choice of low level primitives, the USB drive could be formatted as an FAT32 disk which has the maximum capacity of 4GB, then interrupt 13h could be used to write to/read from its sectors (ah = 03h/02h). Similarly we can use interrupt 14h to initialize communication port (ah = 00h) and send/receive (ah = 01h/02h).

We can also implement a simple driver using in and out instructions. The BIOS stores pointers to the device memory at fixed locations. For example, pointers to all the available serial ports are stored at 0x0400. We can directly drive the serial port by using out instructions after configuring the device using the BIOS provided memory map.

Regardless of our choice of an output mechanism, an important issue is how to protect the disk from being tampered with or detected by the malware. An external disk would appear to be unmounted and we could try to remove BIOS provided memory map from the guest virtual memory. We could use a similar technique to hide the serial port from the guest. If a 0x0 pointer is stored in the BIOS map, the system will interpret it as there not being a hardware serial port available.

Given that we can use Simnow to bind a virtual serial port in the simulator to a real port on the hosting system, we decided to use the serial port for the first stage of our work. Though it is likely to be very slow, it will illustrate a proof-of-concept for our ideas. The same hiding and I/O mechanisms could be ported to an external disk.

2.7.2 Specifying a Watched Process

Our hypervisor is the first layer above the hardware and virtualizes the operating system running on it. We want our layer to be as efficient and as unobtrusive as possible. Thus, we want it to dump the memory and intercept instructions of only those processes which the user deems suspicious and wants to diagnose. In the normal mode of processing, the hypervisor will have all intercepts disabled. It will not watch any memory or system calls and the system should run without considerable performance

penalty.

To let the user start a process with the hypervisor watching enabled, we need to modify the Linux kernel using a loadable kernel module. We evaluated two techniques for doing this: VMPCALL with CR3 address and kernel updates to the task state segment (TSS).

The simplest mechanism is to insert a VMPCALL instruction into the kernel when it creates a new process's page table base pointer. VMPCALL allows the guest operating system to explicitly call into the hypervisor. While this method requires minimal modification to the guest operating system, it does not itself help the hypervisor track sub-process executions.

We use the hardware task switching provided in the AMD architecture [2] to both track watched processes and keep watching any sub-process executions. On each process switch, the kernel updates some fields of the TSS (Task-State Segment) so that the corresponding CPU's control unit may safely retrieve the information it needs [4]. There are some reserved/unused bytes in the AMD TSS. For the process of interest, the Linux kernel does the following:

1. It flags one unused byte, say watchFlg, in the TSS. It is the kernel's responsibility to continue flagging all child processes forked by the malicious one.
2. It explicitly calls VMEXIT with an error code which tells the hypervisor that it has to turn on instruction intercepts. This is done just once and not for all the child processes invoked. The hypervisor enables instruction interception on receiving this trap.
3. Once the malicious process and all its children have exited, it again calls VMEXIT with an error code, which tells the hypervisor to stop tracking. The hypervisor disables all instruction intercept on receiving this trap.

For the while, the hypervisor tracking is enabled, it checks if watchFlg is set on each MOV TO/FROM CR3 and task switch. If it is set, it performs full tracking. If it is not set, it enables just MOV TO/FROM CR3 and task switch interception.

For simplicity, we decided to use the VMPCALL with CR3 address mechanism to specify a watched process. It requires minimal modification to the guest operating system and when combined with fork system call tracing and CR3 inference it is equally powerful as TSS tracking. Furthermore, TSS tracking requires more trust in the guest operating system, which we wish to avoid as much as possible.

3 Implementation

3.1 Enhancing TVMM

The original version of TVMM, on which we based our code, and the Simple Operating System (SOS) that came with it are highly coupled. SOS simply prints a message to the screen and passes control back to TVMM where it would halt. While TVMM was a useful proof-of-concept for understanding AMD SVM virtualization, it was not full featured enough and sufficiently free of bugs to run a more complex OS or program. We modified TVMM in a number of ways to make it boot a general purpose program that was designed to run on bare hardware. As a first step, we modified TVMM to run GRUB Invaders, a multiboot compliant kernel game [27]. In so doing, we prepared our VMM to implement more advanced malware analysis techniques on a general purpose operating system.

3.1.1 Debugging Mechanisms

TVMM has very basic support for debugging. It only allows printing information to the screen which is not reliable since the output could be overwritten by the guest operating system. It was not possible to stop in the middle of the program and examine its state. We developed a framework which allowed us to send debugging data to a serial port and set breakpoints at any given point in the execution of the program. The breakpoints were created by reading memory at a special location (0x00FFFFFF) and setting memory reference breakpoints in Simnow debugger. These debugging mechanisms were very effective. They helped us to detect bugs and errors and to successfully implement various changes to TVMM.

3.1.2 Memory Layout and Management

Originally, the host (TVMM) allocated 4MB of space in the low memory for itself. The guest (VM) was assigned another 4MB after the physical memory of the host. TVMM also sets up one allocation bitmap to manage its memory allocation as we show in Figure 2.

We totally redesigned TVMM's memory structure. In MAVMM, the guest is located in low memory, with the VMM occupying the last few megabytes of physical memory. This approach makes it possible to do identity mapping between guest physical and host physical addresses simplifying access to I/O. After the guest physical memory allocation, we store the code of the VMM followed by two allocation bit maps: one for the host and one for the guest. This allows us to protect host memory from the guest by separately managing their memory. We also use the guest allocation bitmap to allocate and manage memory and structures to be passed to the

guest. These structures include the guest's initial IDT, GDT and paging tables (if needed). We also enhanced the general purpose allocator in TVMM to use a heap. Figure 3 shows the modified memory structure.

3.1.3 Guest Operating Modes

Initially, TVMM only supported booting the guest in 64 bit mode (long mode) with a guest paging structure of 2MB pages. This is inflexible since we also want to be able to boot 32 bit OSes and programs which may expect to start in real or legacy protected mode with paging off. For example, the ELF image of Linux that we tested against would crash at boot using TVMM. It crashed when trying to clear the PE (protected mode) flag of CR0. This happened because the PG (paging enabled) flag, which is required by long mode, was set in TVMM and clearing PE while leaving PG set would create an inconsistent CPU state and a trap.

We modified TVMM to support various different guest operating modes that are configurable at boot time, including real mode, protected mode with paging disabled, protected mode with 4MB memory pages, protected mode with 4KB pages, and long mode with 2MB pages. The original version of TVMM with long mode only created page tables for 4MB of guest physical memory, so we expanded it to support arbitrarily large guest physical memory. Since segmentation is disabled in long mode, TVMM did not handle properly setup the CPU segmentation registers. To support legacy protected mode, we had to set up the guest's GDT table. Basically, we created a minimal number GDT entries: a single code segment and a few data segments. Each segment has 4GB range and is based at virtual address 0. Memory management is by far the most complicated task that our VMM has to handle, due to the fact that there are many different CPU and paging modes.

The VMM itself runs in long mode with a simple mapping of the first 1 GB of both lower and higher virtual memory to the first 1 GB of physical memory. It loads the paging structure and GDT values statically from within real mode and then jumps directly into long mode. This is largely unchanged from the original version of TVMM.

3.1.4 Hiding Host in Memory Map

Not all regions of machine physical memory are available to be used by the operating system. Some memory is reserved by the BIOS and some is used for I/O mapping, etc... The information about which regions are available and which are reserved is provided in a structure called e820 memory map, which can be queried by using function number 0xE820 of BIOS interrupt 15h. TVMM is

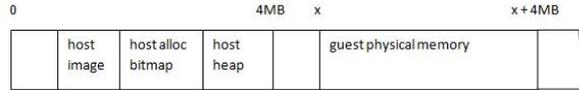


Figure 2: Original TVMM Memory Layout

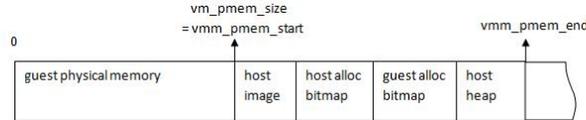


Figure 3: MAVMM Memory Layout

multiboot compliant and uses the GRUB boot loader to ease several tasks including locating itself and the guest image on disk, and querying the physical memory structure. GRUB loads the guest image into memory as the first kernel module and passes its location to TVMM via a structure called multiboot information. This structure also contains an e820 memory map.

MAVMM is also a multiboot-compliant boot loader for its hosted guest VM. To hide the existence of the VMM from the guest, we modify the multiboot information structure to reserve the VMM memory. Since we provided identity mapping of guest physical and host physical memory, we added a function which marks the memory regions that the host uses as reserved in the e820 map before passing it to the guest. This gives the guest the illusion that these regions are used by the BIOS and are not available for the OS.

3.1.5 Interrupts

Interrupt handling is also complicated by the state and mode of the CPU. Interrupt handling is fairly similar in real mode and protected mode, except that in real mode, idtr (the interrupt descriptor register) points to an Interrupt Vector Table (IVT) and in protected mode idtr points to an Interrupt Descriptor Table (IDT). Each IVT entry is 4 bytes and each IDT entry is 8 bytes in size. TVMM did not set up initial interrupt handlers for the guest since it expect the guest to set up its own IDT. But this might break OSes which boot in real mode and rely on BIOS interrupts for bootstrapping (e.g., the Linux kernel boot sector uses BIOS interrupts to read from the disk). To prevent this from happening when the guest boots in real mode, we set the guest IDTR to point to BIOS interrupt vector table which begins at memory address 0, with a limit of 0x3FF. In other cases, we set the idtr's limit field to 0 so that the guest know that it should set up its own IDT if necessary.

3.1.6 Booting Different Types of Guest Images

The original version of TVMM only supported booting ELF guest images. To make MAVMM more general purpose, we configured it to be able to also boot the multiboot format and the Linux kernel boot format. This allows us to boot standard Linux kernel images as well as the ELF image. We also needed standard multiboot compliance to boot Invaders. We allow the user to specify the image type at boot time (using GRUB).

The multiboot loader checks the multiboot header checksum and then loads the image at 0x10000 for real mode and 0x100000 for protected mode. For historical reasons, the Linux kernel boot process contains many static addresses and fields [4]. Our Linux kernel loader reads the boot header to determine the number of sectors in the setup segment (the real mode kernel code) or it defaults to 4 if left unspecified by the kernel (another historical default). It then copies the setup segment code to address 0x90200. Next it copies the remaining protected mode kernel segments to 0x10000 if the kernel is in zImage format or 0x100000 if the kernel is in bzImage format. The loader then sets the guest entry instruction pointer to 0x90200.

3.1.7 Handling Guest State

One of the final enhancements we made to TVMM was to properly save and restore the state of the CPU on VMEXITS. Using the original version of TVMM, we were not able to play the Invaders game inside the VMM because any key press caused a crash. After fixing the interrupt handling (described before), key presses were still crashing due to a base pointer address from the host being loaded inside the guest. We spent considerable time debugging this problem, which was complicated by the fact that the reproducible error actually happened inside the BIOS interrupt service routine not inside of the code of Invaders itself. This problem was present in a variety of configurations of IDT, paging modes, and operating

modes.

We finally discovered that TVMM was not properly implementing SVM virtualization support on a VMRUN/VMEXIT by loading and saving guest state. This caused the guest to be returned in a state different from what was there before VMEXIT, and the guest's behavior was very erratic and undefined. The hardware virtualization support automatically performs some state manipulation in the vmcb on a VMRUN or VMEXIT, but it is up to the VMM developer to ensure that all the state is saved and restored when transitioning between the host and the guest. The SVM extensions added the VMLOAD and VMSAVE instructions to the ISA to assist this, but they only load and save internal CPU state that is not normally accessible to the programmer. The VMM developer has to handle some other *visible* states like the rbp, rbx, rcx, rdx, rsi, rdi and r8-r15 registers. We added code which stored these values after a VMEXIT and restored them when the VMM transferred control back to the guest. We also added VMLOAD and VMSAVE instructions around the VMRUN instruction to ensure that the guest's internal state is properly preserved.

3.1.8 Development Experiences and Discussion

As we have discussed, we enhanced TVMM and fixed numerous bugs. In the process, we thoroughly understood the code of TVMM and documented it heavily with comments. We hope that this enhanced version will be of value to researchers in the future.

In parallel to our work enhancing TVMM to support Invaders, we also worked on booting Linux. We used the mkelfImage utility to make the ELF image. It creates a stub loader that prepares the ELF image for loading and then transfers control to Linux. We were able to prepare the state of the guest enough that the stub loader completed and transferred control to Linux. The kernel then began booting, but crashes fairly early on in the boot process. We have not yet been able to debug the problem, but we are confident that once we set up the guest with the state that Linux is expecting, we will be able to boot Linux. Unfortunately, in review of the assembly code that performs the first few stages of the Linux boot process and in looking at the code for other boot loaders, Linux has some odd and very particular assumptions (many of them historic) about the initial state of the system that we must ensure we provide in the guest. We found that when directly booting the Linux kernel image (using the boot loader we described above), we also ran into similar problem with machine state that Linux did not expect.

3.2 Feature Extraction

3.2.1 Memory

Extracting process memory is dependent on which paging mode the guest OS uses. For example, an OS can use real mode or paged mode addressing. If paged, it can use either long or legacy mode addressing. Each addressing mode has different number of page tables and thus different ways of interpreting a virtual address. If the guest uses four level nested paging, our hypervisor has to do two levels of address translation manually, similar to that done by hardware when guest is running. The VMCB has fields for the paging registers for both host and guest which the hypervisor can inspect. According to the addressing mode, it parses the virtual address, uses the nested paging CR3 (nCR3) and guest CR3 (gCR3) values saved in the VMCB to get the final physical address.

Invaders does not use paging natively (though we also modified it to run in long mode with minimal changes). Extracting its memory involves just the host level page tables pointed to by nCR3. We read the page tables and traverse to the page table entry for the address we wish to extract. At the lowest level of page table, we check if an entry has been accessed or modified. If so, the entire page is read from physical memory and the bytes are sent over the serial port.

In order to gather parameters during system call and network call tracking, we use a similar translation. The guest physical address is viewed as a 64-bit host virtual address. Parts of this address are used as offsets into the four nested page tables (pointed by nCR3) to get the physical address. The size of the buffer to be read is specified as a system call parameter and only those many bytes are fetched from the memory.

3.2.2 System Calls

Implementing system call tracking has been straightforward. On int 0x80 intercept, MAVMM reads the eax/rax register (depending on if the guest is in long mode) to get the system call number. We used the system call struct sysent table used by the strace Linux implementation to identify each system call by the system call number. We only use the system call name and number of parameters fields of struct sysent. We index into this table and send the corresponding string to the serial port. MAVMM reads the parameters which reside in registers (ebx, ecx, edx, esi, edi, ebp) and sends them to the serial port as well.

Table 1: Time to Dump a 2MB Page

Time Units	Mean	Standard Deviation
clock cycles	487707509	354
ms	542	4e-3

3.2.3 Serial Port Communication

We used the serial port interface provided by the Simnow simulator to test the serial port communication. Simnow provides the abstraction of a serial port by creating a named pipe. We configure such a serial port in one terminal and using the same named pipe in another terminal, we see communication between the two.

We wrote a function *outf* which accepts formatted strings like `printf` and sends them to the serial port. We write to and read from the serial port using simple IN, OUT assembly instructions. COM1’s base address is found at physical address 0x0400, COM2’s base address at 0x0402 and so on. So we can determine which COM port is present and we set up the Simnow serial port connection accordingly. We are also able to configure the serial port to various speeds and options to better enable the analysis data to be extracted. While we use a named pipe for serial communications within Simnow, the code that interacts with the serial port is not dependent on Simnow and should work on bare hardware with a real port.

4 Evaluation

We were not able to boot a general purpose operating system with our VMM yet, so we had to approximate the evaluation of our system. Our evaluation setup consists of an AMD Simnow virtual machine with a 900Mhz processor and 256 MB of RAM. The simulator ran on a 2.6 GHz AMD Ahtlon processor with 4 GB of RAM and Ubuntu Linux 2.6.22-9.

4.1 Data Extraction

Though the serial port is not the most efficient mechanism for transferring large amounts of data, when used inside Simnow, it was faster than we expected. We instrumented TVMM and an external reading process to compute the time needed to dump a single 2 MB page over the serial port. We configured the serial port to be in the fastest mode and disabled interrupts. The results are shown in Table 1. While it is still costly to dump a single page, we can envision spreading the work of dumping the memory across many VMEXITs. We can also store the memory in a copy-on-write manner and dump it all after the malicious process exits. This will prevent any large timing discrepancies detectable in the guest.

Table 2: VMSCALL Overhead

Time Units	Mean	Standard Deviation
clock cycles	2262	533
μ s	2.5	0.6

Table 3: VMM Detection

Detection Technique	VMWare	Xen	MAVMM
Red Pill (IDT Check)	YES	YES	NO
LDT Check	YES	YES	NO
VMWare Port	YES	NO	NO
MSW Check	YES	NO	NO

4.2 Virtualization Overhead

We added code to Invaders that explicitly causes a trap into the VMM. We use the VMSCALL instruction to send data in the `eax` register to the VMM. This trap is similar to a null procedure call into the VMM and approximates the overhead of any VMEXIT switch. We used the `rdtsc` instruction to query the program counter before and after issuing the VMSCALL instruction. We show the results in Table 2. We found that the trap into the hypervisor consumed 2.5 μ s. Because the `tsc` is properly emulated by Simnow, this overhead is likely to be comparable to that of a real machine.

4.3 Detectability

The last set of evaluations we performed on MAVMM was to implement some well-known VMM detection algorithms inside of Invaders. We implemented the red pill technique which looks for an IDT in high memory [24]. We implemented a heuristic developed by Quist and Val-smith that checks the local descriptor table LDT register value [29]. We implemented a check for the I/O port that VMWare uses to talk to its guest. Finally, we implemented a check for a smaller value returned from the machine status word indicating the presence of a VMM. We ran each of these techniques inside of VMWare, para-virtualized Xen 3.1, and MAVMM (see Table 3). MAVMM was not susceptible to any of these attacks while Xen and VMWare were susceptible to some or all of them. A TLB sizing attack as described in [25] can be used to detect MAVMM but it is a heavy weight test that consumes considerable CPU resources to reliably detect the VMM. We do not think this is practical for a stealthy malware application to use without being easily detected.

4.4 Experiences and Discussion

Using our MAVMM prototype, we were able to dump the physical memory of Invaders and inspect the location of the alien ships on the screen outside the VMM. While Invaders is in no way malware, we were able to prove that we can introspect upon an application running inside the VMM and access information that would have otherwise been inaccessible. We were also able to intercept interrupts to determine when the user presses a key on the keyboard. We can use this for example to make a full report of all the keys pressed during a single session of the game. Again, this information is not useful by itself but proves that we are able to extract information from the running guest. In future work, we plan to more fully evaluate our extraction techniques against Linux and eventually Windows.

5 Related Work

VM Introspection (the process of examining a process inside a virtual machine from *outside* the virtual machine) was introduced by Garfinkle and Rosenblum [11]. While other work leverages this idea for security purposes like process tracking [15], intrusion detection [30, 20], and malware detection [14], our work focuses on *hardware-supported* introspection. We also use an entirely new VMM while previous work modified Xen or VMWare.

A significant motivation for our work was prior work on malware analysis in a non-virtualized environment. Malware has proven to be very deft at avoiding analysis through debugger detection [12], fingerprinting [19], and code obfuscation [23]. Operating system based systems like Saffron [23] and simulator based systems like Renovo [17] are able to analyzing running malware and automatically remove packing or encryption. The Janus project studied system call tracing for program understanding [9]. We aim to implement a subset of this previous work on malware analysis inside our VMM.

Because of the danger of running malicious software, virtualization offers strong protection and rollback mechanisms to aid live debugging. Because virtual machines are so commonly used by malware analyzers, virtualization detection is becoming part of modern malware. The attacks range from simple IDT based attacks [24, 18] to complicated TLB sizing attacks [10]. We believe it is reasonable to assume that any virtualization platform will introduce some detectable changes to the guest system, so we aim to provide *minimal* detectability for the VMM using hardware virtualization. Malicious software must firstly evade detection and removal so that it may carry out its task. Since preventing analysis is a secondary task, the malicious software is unlikely to go to great lengths to avoid analysis if by doing so it in-

creases its detectability.

Finally, we have found that general purpose VMMs are as large and complex as a modern operating system. It is likely vulnerabilities more vulnerabilities will be discovered in these systems that will make them impractical for malware analysis because of the loss of strong isolation. Consider the following two vulnerabilities in VMWare and Xen that can both lead to compromise of the host system from inside the guest: CVE-2008-0923, CVE-2008-0600 [5].

6 Conclusion

Our design and implementation of MAVMM shows that a very lightweight hypervisor, custom built for malware analysis, is feasible. We investigated various features which are necessary for malware analysis, and different strategies to implement these features in a VMM. We also built a simple malware analysis supported VMM. At this stage, our VMM - MAVMM - is able to boot and run a multiboot compliant kernel game, GRUB Invaders, unmodified and extract simple forensic information about its activities. As MAVMM uses hardware virtualization extensions it remains fairly undetectable, its memory is protected from both DMA and the guest operating system code. VMMs are becoming increasingly large and complex and are said to have OS like exploitable vulnerabilities. Being lightweight and simple, MAVMM also remains less vulnerable.

The TVMM research code that we started with was inflexible and had a number of flaws. TVMM can essentially boot only its 'sample operating system'. Booting a real OS or a general purpose program requires myriad changes and proved to be a daunting task. MAVMM has been extended to accommodate different guest operating modes. It can handle different paging modes, page sizes, CPU operating modes (real/protected/long) and several types of guest images.

An important goal that we started with was to provide the Computer Science research community with a simple and easy to enhance virtualization platform. MAVMM provides that platform, and by being simple (around 2000 lines of code) and well documented MAVMM makes it easy for other researchers to add new functionality to it, or modify it to serve their purposes.

References

- [1] AMD. Amd simnow™ simulator. <http://developer.amd.com/tools/simnow/Pages/default.aspx>.
- [2] AMD. Amd64 architecture programmer's manual volume 2: System programming.
- [3] AMD. AMD64 Virtualization Codenamed *Pacifica* Technology: Secure Virtual Machine Architecture Reference Manual. Tech. rep., May 2005.

- [4] BOVET, D. P., AND CASSETTI, M. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.
- [5] CVE. Common Vulnerabilities and Exposures. <http://cve.mitre.org/>.
- [6] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.* 36, SI (2002), 211–224.
- [7] ECONOMICS, C. 2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets, and Other Malicious Code. Tech. rep., June 2007.
- [8] F-SECURE. Agobot. www.f-secure.com/v-descs/agobot.shtml.
- [9] GARFINKEL, T. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).
- [10] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)* (May 2007).
- [11] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Inspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).
- [12] HOLZ, T., AND RAYNAL, F. Detecting honeypots and other suspicious environments. *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC* (15-17 June 2005), 29–36.
- [13] INTEL. Intel®Virtualization Technology Specification for the IA-32 Intel®Architecture. Tech. rep., April 2005.
- [14] JIANG, X., WANG, X., AND XU, D. Stealthy Malware Detection through VMM-Based "Out-of-the-box" Semantic View Reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 128–138.
- [15] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), USENIX Association, pp. 1–1.
- [16] KANEDA, K. Tiny virtual machine monitor. www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/, June 2006.
- [17] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malware* (New York, NY, USA, 2007), ACM, pp. 46–53.
- [18] KLEIN, T. Scooby Doo - VMWare Fingerprint Suite, 2003. www.trapkit.de/research/vmm/scoopydoo/index.html.
- [19] KORTCHINSKY, K. Honeypots: Counter measures to vmware fingerprinting, Jan 2004. <http://seclists.org/honeypots/2004/q1/0015.html>.
- [20] KOURAI, K., AND CHIBA, S. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (New York, NY, USA, 2005), ACM, pp. 197–207.
- [21] LTD, A. K. S. Security Statistics, 2008. <http://www.esafe.com/csrt/security-statistics.aspx>.
- [22] NAVARRO, P. G. M. gtrace: function call and memory access traces of dynamically linked programs in IA-32 and IA-64 linux. Tech. rep., November 2002.
- [23] QUIST, D., AND VALSMITH. Covert Debugging: Circumventing Software Armoring. Blackhat Las Vegas, August 2007. www.offensivecomputing.net/bhusa2007/dquist-valsmith-covert-debugging-paper.pdf.
- [24] RUTKOWSKA, J. Red Pill: Detect VMM using (almost) One CPU Instruction, November 2004. <http://invisiblethings.org/papers/redpill.html>.
- [25] RUTKOWSKA, J., AND TERESHKIN, A. Blue Pill – Is-GameOver(), Anyone? Blackhat Las Vegas, August 2007. <http://bluepillproject.org/>.
- [26] SA/NV, D. IDA Pro Disassembler and Debugger. www.hex-rays.com/idadpro/.
- [27] THIELE, E. Invaders. <http://www.erikyyy.de/invaders/>.
- [28] TIS COMMITTEE. Tool interface standard (TIS) executable and linking format (ELF) specification, May 1995. Version 1.2.
- [29] VALSMITH, AND QUIST, D. Hacking Malware: Offense is the new Defense. Defcon 14, August 2006. www.offensivecomputing.net/dc14.
- [30] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.* 39, 5 (2005), 148–162.
- [31] WAHLIG, E., AND HUANG, W. Amd barcelona and nestedpaging support in xen, 2007. http://xen.org/files/xensummit_4/2007XenSummit-AMD-Barcelona_Nested_Paging_WahligHuang.pdf.
- [32] WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. T. Automated web patrol with strider honeymoons: Finding web sites that exploit browser vulnerabilities. In *NDSS* (2006).
- [33] Wireshark version 0.99.6: A network protocol analyzer for windows and unix. www.wireshark.org.
- [34] YUSCHUK, O. Ollydbg. <http://www.ollydbg.de/>.