MAVMM: Lightweight and Purpose Built VMM for Malware Analysis

Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King
Department of Computer Science
University of Illinois at Urbana-Champaign
{*anguyen7, nschear2, jung42, godiyal2,kingst*}*@illinois.edu*

Hai D. Nguyen
Hanoi University of Technology
haind93@gmail.com

***Abstract*—Malicious software is rampant on the Internet and costs billions of dollars each year. Safe and thorough analysis of malware is key to protecting vulnerable systems and cleaning those that have already been infected. Most current state-of-the-art analysis platforms run alongside the malware, increasing their detectability. This reduces the value of analysis because some malware is known to behave differently when being analyzed. Virtualization offers a compelling platform for malware analysis, with strong isolation and the ability to save and restore guest state. Current virtual machine monitors (VMMs), however, are not designed for malware analysis. Due to their complexity, they often fail to provide transparency and even expose vulnerabilities which could be exploited by the malware running inside guest system.**

We propose a lightweight VMM (namely MAVMM) that is designed specially for a single job: malware analysis. MAVMM does not implement unnecessary virtualization features commonly found in general purpose hypervisors, including virtual device emulation. We take advantage of hardware virtualization support to make MAVMM more simple, secure and transparent. In this paper, we describe the design and implementation of MAVMM, and the features that we can extract from programs running inside the guest OS. We evaluate our platform in three aspects: functionality, detectability and performance. We show that our system can extract useful information from malicious software, and that it is not susceptible to known virtualization detection techniques.

***Keywords*-Malware analysis; virtual machine monitors; security;**

I. Introduction

Malware–a representative term for viruses, worms, spyware, trojan horses, adware and rootkits–is a major threat to today's highly connected computing environments. Annual damage from malware is estimated to be more than 10 billion dollars [11], more than 40 percent of companies worldwide report business disruptions due to malware [7], and 55% of all online users believe their systems had been infected [9]. All signs point toward malware becoming a more significant threat in the future.

Malware analysis plays a crucial role in countering this trend. Through detailed analysis of a particular malicious application, security researchers will be able to gain insight into its intention, its runtime behaviors, and the risk that it creates. This knowledge is very valuable in predicting the threats posed by the malware, creating appropriate 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 [36], debuggers [48], and black box analysis such as function call tracing (e.g., strace) and network sniffers [8]. While these methods are useful to some extent, each suffers from certain drawbacks. Disassembling, like other static analysis techniques, can be circumvented by packing or dynamic code translation [26, 43]. Dynamic black box analysis only gives an incomplete view of the malware's behaviors. Debugging, on the other hand, provides a more exhaustive view but is vulnerable to debugger fingerprinting [42, 19]. As malware gets more and more complex, it is often impractical and unnecessary to analyze each and every instruction.

Another common approach in malware analysis is to deploy analyzing tools in conjunction with virtualization technology, taking advantage of its strong isolation, and its ability to take snapshots and roll back the guest's state. In addition, VMM-based analyzers have a unique ability to monitor virtual machine based rootkits [27, 35]. Commodity virtual machine monitors (VMMs) such as Xen and VMWare have already been used in malware analysis [51, 10]. Unfortunately, we are facing an advanced and intelligent enemy. Malware writers have deployed increasingly complex techniques to evade detection and prevent forensic analysis, using side channels [34] or artifacts of the virtualization platform [29, 13]. When the malware detects it is running inside a virtual machine, it often exits to prevent further analysis. Recent malware, such as the infamous Storm and some versions of Conficker are known to behave in this fashion [49, 50]. Other types of malware even try to act differently to fool analyzers of their intention [12] Therefore, detectability of the virtualization platform will greatly affect accuracy of the analysis system. General purpose VMMs, including Xen and VMWare, are inherently not suitable for this task. They are designed for functionality and performance, not transparency. As one example, to support virtualization of multiple virtual machines running multiple guest OSes at the same time, these VMMs need to implement virtual device emulation. This device emulation usually leads to the inclusion of a host OS with millions of lines of code inside their trusted computing base (TCB). Due to their complexity, commodity VMMs often fail to provide transparency. A recent study by Garfinkel *et al.* [17]

shows that device emulation is the main source of logical and timing discrepancies between virtualized and non-virtualized environment. Putting detectability aside, these features also expose many vulnerabilities that could be exploited by the malware under analysis to escalate privileges, DOS the analysis platform, or bypass security restrictions. As an evidence, there have been at least 17 known vulnerabilities in Xen 3.x, 42 in VMware Workstation 6.x, and 165 in VMware ESX Server 3.x [37, 38, 39]. If the VMM layer can be easily compromised, it becomes very risky to trust the analysis platform based upon it.

In this paper, we propose the architecture of MAVMM, a VMM specially built for malware analysis. By taking advantage of hardware-support for virtualization [3, 21] and focusing only on malware analysis functionality, we were able to keep MAVMM small and simple. The TCB of our system is 2 to 3 order of magnitude smaller than other VMM-based malware analysis platforms. Our goals for this work are:

- Ability to extract useful data for malware analysis.
- Minimum trust in the guest OS.
- Simplicity and compactness for the VMM, which improves transparency and security.

MAVMM works by extracting runtime analysis data, which we will refer to as *features*, from the monitored guest application. The features that we extract from the guest OS include both fine-grained information and high-level information: execution traces, memory dumps, system calls, disk accesses, and network interactions. These features can then be used by malware analyzers to create a fairly complete picture of the malware.

While complete undetectability is most likely a panacea [17], our system is qualitatively more difficult to detect than simply running analysis tools alongside the malware or using a commodity virtualization system. In addition, our experiments show that common methods used to detect virtualization are ineffective against MAVMM. Our main contributions can be summarized as follows:

- We propose a more transparent and secure malware analysis architecture, using a purpose-built VMM and hardware virtualization support.
- We implement a prototype system, demonstrate that MAVMM can extract useful data, and that common VMM detection techniques are ineffective against it.
- We open the source code of our VMM and give other researchers access to it. Beside malware analysis, this simple hypervisor with hardware supported VMM introspection will be useful for auditing, logging & replaying, and many other purposes. Our code is accessible at mavmm.sourceforge.net

The remainder of this paper is organized as follows. We present the general design of MAVMM in Section II and describe specific implementation details in Section III. In Section IV, we present the result of our evaluations. We further examine related works in Section V and conclude with Section VI.

II. MAVMM Design

To develop our architecture, we study various techniques for virtualizing the system, extracting analytic features from the guest and communicating with the analysis platform. In this section, we present the high level design of MAVMM and explain our design decisions. Our design is independent of virtualization platform (AMD SVM/Intel VT) and guest operating system (OS). We describe implementation related details, that are specific to AMD SVM and Linux, in Section III.

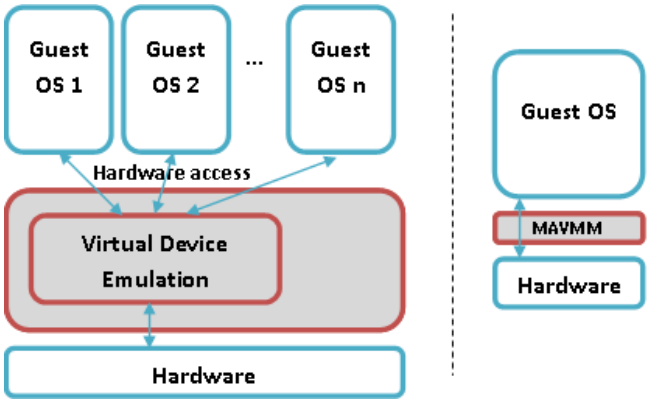


Figure 1. Comparison between general purpose VMMs (left hand side) and MAVMM (right hand side). The TCB is shaded. MAVMM lets most hardware access requests go through without interception.

A. Hardware Virtualization Technology

Both AMD and Intel currently offer hardware virtualization support in their processors, including the mainstream Intel Core2 Duo and AMD Opteron. Hardware virtualization provides faster virtualization performance, and several features to simplify VMM implementations, and therefore is a natural choice for MAVMM. For the purpose of malware analysis, our hypervisor mainly takes advantage of the following features offered by hardware virtualization [3]: an additional CPU mode for the hypervisor, nested paging, address space identifiers (ASID), an IOMMU, and event interception and injection.

B. Special Purpose Hypervisor

One of our primary goals is to keep the hypervisor thin and lean, as we believe simplicity will aid transparency and security. Even though commodity VMMs, such as Xen, KVM or VMware, use hardware virtualization, their code bases are still too large and complex for our purpose. All of them include a general purpose OS inside their TCB, and implement many virtualization features that are not necessary for malware analysis. Needless to say, this complexity

is a fruitful land for software bugs. A recent study shows that programs usually contain between 6 and 16 bugs per 1,000 lines of executable code [5], while another puts the number at 2 to 75 depending on module size [31]. Indeed, there are known attacks that break VMM sandboxes from the guest environment, allowing attackers to take control of the hypervisor and the host OS [37, 38, 39, 47]. This situation is clearly undesirable in VMM-based malware analysis. Once the malware has taken control of the hypervisor, it is very likely that it will find a way to break into the analysis platform. This observation led us to the most important design decision in this project: *design a new, special purpose VMM* instead of instrumenting a commodity VMM with malware analysis capabilities. Unlike traditional VMMs, MAVMM lets the guest interact directly with hardware for most of its operations. The VMM only makes interception at a few places, in order to protect its integrity and log the guest's behaviors for later analysis. A comparison between MAVMM and general purpose VMMs is shown in Figure 1.

C. Boot-strapping the Hypervisor

To get an accurate view of the monitored system's activities, MAVMM needs to start earlier and run at a higher CPU privilege level than the software under analysis. Thus, we decide to boot MAVMM directly from a boot loader. Another option is to run our VMM on top of or alongside a host OS, similar to Xen [4], KVM [20], and VMWare Workstation [45]. The higher level of abstraction provided by a host OS would make it easier to bootstrap the VMM, however we choose to avoid it to keep our platform small and simple.

D. Protecting Hypervisor Memory

MAVMM uses nested paging to protect its memory from being tampered by the guest. Nested paging adds an additional layer to the hardware address translation process. This process is illustrated in Figure 2. From the perspective of the guest, guest physical addresses are the same with hardware addresses of RAM chips. In reality, they are translated into host physical addresses with assistance from a newly inserted nested page table (NPT). By setting up the NPT appropriately, MAVMM can redirect guest requests to access its memory region, and hide its existence. From this point, we will refer to this region as the *VMM region*.

Paging redirection protects MAVMM from memory access by the CPU, but it does not protect MAVMM from direct memory access (DMA). To keep our hypervisor from being tampered with by external device DMA, we use the IOMMU offered by hardware virtualization. IOMMU allows flexible control of each device's view of the main memory. This is done by using a translation table, to control the mapping from device virtual addresses to memory physical addresses.

E. Feature Extraction

This section describes the information we glean from a running VM, our techniques for exporting the data, and an optimization for avoiding logging unnecessary states and events.

1) Features: It is important that MAVMM can extract useful data in addition to running the malware safely and invisibly. We support extraction of the following features from applications running inside the guest: fine-grained execution trace, memory page, system call, disk access, and network access. These features are the fundamental blocks upon which other analysis functions could be implemented.

An execution trace provides the highest level of detail, similar to what a run-time debugger can achieve. This information plays an important role in understanding the malware's internal operations. To get the execution trace of a guest program, we single step through it's execution and record each instruction. We do so by virtualizing the TF flag within *rflags* register and set it to 1. This would create an #DB exception, which could be intercepted by MAVMM, after every guest instruction. We keep some state to learn whether the guest or our VMM raised the TF flag. #DB exceptions created by the guest should be forwarded to it, while the other ones need to be processed transparently.

When intercepting events, such as system calls and network accesses, MAVMM fetches guest pointers from memory. These pointers contain guest logical addresses, and MAVMM needs to translate them into host physical addresses before accessing the data that they point to. To translate a guest logical address to the corresponding host physical address, we duplicate functionality of the segmentation unit and the paging unit in software. Using the guest's segmentation and paging structure, MAVMM can translates a guest logical address to guest physical address. Because we use an identity map in our nested page table, this guest physical address and its corresponding host physical address are the same[1]. With this translated host physical address, MAVMM is able to read the data from memory for further processing.

System calls are the main interface for software to interact with and make changes to system state. A log of executed system calls is often good enough to get a rough idea of what the malware is trying to do. MAVMM provides the ability to record all system calls that a guest program invokes.

Most malware tries to gain access to network, either to propagate itself to other hosts (worm), send out stolen data (spyware), or contact the master for further instructions (bot). To remain persistent, malware often needs to make changes to the hard disk. Therefore, network and disk monitoring are crucial features. Using system call interception, MAVMM can track network accesses and disk accesses as they happen.

[1]this is true except for VMM region

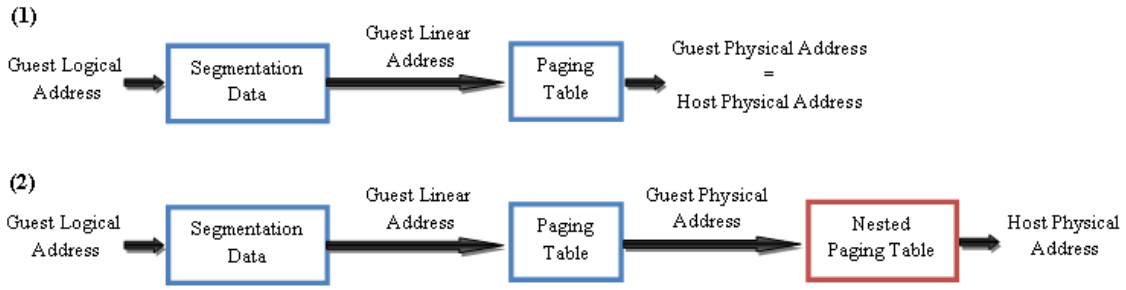


Figure 2. Address translation process for a guest OS without nested paging (1), and with nested paging (2).

2) Getting Analysis Data: One challenge that we face in our design process is how to get the data logged by MAVMM out of the monitoring system, so that more analysis could be done upon it. The key problem is that MAVMM allows the guest to retain direct control of devices, so accessing I/O safely and invisibly requires careful design.

We have several choices for extracting data from our analysis platform: use the same hard-disk as the guest OS, use a separated hard disk, use an USB flash drive, or use a system port such as the serial port. We decide to avoid using the same hard disk with the guest system, to minimize detectability and avoid possible contentions with the guest OS. We do not want to use guest drivers to perform our I/O because, if compromised, the guest could alter our analysis data. This leaves external USB drive and serial port communication as the preferred methods for extracting data. For both external drive and serial port, we can use BIOS services to dump the data out. We can also implement a simple driver to access serial port devices directly without using BIOS services.

3) Selective Analysis: We want the hypervisor to be as efficient and as unobtrusive as possible. Thus, we enable full analysis capabilities only when necessary. For example, analysis should be disabled when the guest OS is booting in clean state. As a result, MAVMM has two operating modes: *compact* and *full*. In *compact* mode, the hypervisor has most interceptions disabled and the monitored system runs without considerable performance overhead. It only keeps activated a few interceptions that are absolutely necessary for VMM protection. In *full* mode, however, MAVMM intercepts and extracts all features mentioned in Section II-E1.

MAVMM can selectively monitor specific processes and ignores other unimportant ones. This design removes unnecessary noises from the log and accelerates overall analysis process. To monitor selective processes, MAVMM needs to be notified each time a process switch takes place. It also needs a mechanism to identify the next process that is going to be executed. In most modern operating systems, each process has a separate virtual address space, and a different paging structure. With support from hardware virtualization, MAVMM intercepts any attempt to write to the paging base pointer[2]. This allows our VMM to take control during process switches, right before the incoming process get executed. MAVMM then uses VM introspection [18] to get the name or process id of the next process, and compare it with a list of processes that need to be monitored. If the next process is in this list, MAVMM will enable full mode.

To track sub-processes created by an application under analysis, MAVMM infers their names using system call tracing. When a monitored process invokes a system call (e.g., fork) to create a new process, the newly created process will be added to the monitoring list. Similar technique has been demonstrated for tracking processes from a VMM in Antfarm [24].

III. Implementation

In this section, we discuss specific details related to the hardware virtualization technology and guest OS that we have chosen for our prototype implementation. We choose AMD SVM technology mainly because it offers more protection, and use Linux because of our familiarity with this OS. Nevertheless, we believe that our system can support Intel VT and Windows. We plan to implement this support in the future.

Initially, we started with TVMM [25], a small virtual machine monitor built on top of AMD SVM. TVMM was a good starting point and we reuse most of its header files, but it was an incomplete system incapable of booting any real OS. Furthermore, TVMM does not support our analysis features.

A. Hardware Virtualization Technology

We decide to use the AMD Secure Virtual Machine (SVM) extensions for MAVMM. Because the memory management unit (MMU) is on die, AMD engineers are able to offer more advanced virtualization features than the comparable extensions from Intel. AMD SVM natively supports nested paging in hardware. It also provides a convenient mechanism to reserve physical memory from being accessed by DMA. In addition, AMD offers a simulation environment

[2]This pointer is stored in the CR3 register

(AMD Simnow) with many debugging supports, which will be useful for our development and testing.

B. Boot-strapping

We use the GRUB boot loader to start our system. Our VMM executable is stored in a simplified 32-bit ELF format readable by GRUB. We used Xen's mkelf32 utility to build this simplified ELF image from raw object files. When our system boots up, GRUB starts in host mode and begins to load MAVMM. GRUB passes a multiboot info structure to MAVMM, which defines the memory map, command line arguments, and any additional parameters that we specified to GRUB.

Initially, we passed the guest OS image to our VMM using the *module* parameter in GRUB configuration file and tried to execute this image directly from our hypervisor (similarly to TVMM [25]). This approach requires MAVMM to initialize the booting environment to a state as expected by the guest OS, and make sure that it does not overwrite the guest OS image in memory while doing so. This turned out to be very complicated so we changed to a different design. Instead of loading the guest OS directly, MAVMM transfers control back to a second round of the GRUB boot loader. This time GRUB will be started in guest mode to prepare the environment and boot the virtualized guest OS. To do this, MAVMM sets the initial instruction pointer address of the guest to 0x7c00, after it has finished setting up appropriate interceptions and protection mechanisms. 0x7c00 is the beginning address of the loaded master boot record, which contains executable code of GRUB boot loader.

C. Protecting Hypervisor Memory

For simplicity, we create a nested page table and fill it with an identity mapping from guest physical address to host physical address for all memory pages available in the system, excluding the pages used by MAVMM itself. When the guest tries to access this region, a nested page fault (#NP) exception will be raised by the CPU. MAVMM intercepts and handles this fault to hide its existence. We virtualize the VMM region using additional space in an external USB drive. Whenever the guest tries to access (read from / write to) a memory location inside this region, MAVMM executes its request on the additional space provided instead. If guest tries to query this USB port, our hypervisor will intercept and return as if there is no device attached to it.

To protect the VMM from being affected by external device DMA, we use the Device Exclusion Vector (DEV) feature of AMD SVM. DEV is an early version of IOMMU, which allows simple memory protection from DMA accesses. It uses a user-given bitmap to decide which memory pages are available for external DMA. We simply mark the VMM region as unavailable, and set one of the DEVBASE registers to our modified DEV using DEVCTL PCI configuration space function block [3].

D. Features Extraction

1) System Call: Linux applications can invoke a system call in two different ways: by executing the interrupt (int) 0x80 assembly language instruction, or by executing the *sysenter* instruction[3]. Similarly, the kernel can exit from a system call by executing the `iret` or the `sysexit` assembly language instructions [6].

Linux uses the `eax` register to pass a system call number from a user program to the kernel. The user mode process also finds return code of the system call in the `eax` register. AMD SVM allows software interrupt INTn and IRET instructions in the guest to be intercepted using control bits in the Virtual Machine Control Block (VMCB) [3]. During a software interrupt interception, the hypervisor first checks if the instruction is int 0x80. AMD SVM allows us to intercept all software interrupts, but does not provide information on which specific vector number was called at the point of interception. To test whether vector 0x80 was invoked, we fetch current instruction's opcode from the guest by walking through it's paging table in software. If it is indeed 0x80, our hypervisor reads the system call number from `eax` register and process it accordingly. The intercepted software interrupt is then injected into the guest before MAVMM passes control back to it.

For some system calls, such as *sys_read*, useful data is presented only after the handler had finished its execution. To get the data that was actually read from disk, we need to intercept *sys_read* return using IRET interception. We also need to maintain a mapping from the id of the thread which invoked a system call to the system call that it executed. For *sys_read*, this is the pointer to the buffer where its ouput will be stored. This mapping is added when MAVMM intercepts int 0x80 instructions. When an IRET takes place, MAVMM uses VMM introspection to get the ID of current thread. It then looks up information about the system call that this thread invoked. If it is a *sys_read*, data located at the receive buffer along with the returned buffer size will be logged.

In cases where both the CPU and Linux kernel can support sysenter/sysexit instruction, the libc wrapper function may use them to invoke system calls, as they are faster than INT and IRET. Intercepting sysenter/sysexit is a bit more complicated since it is not directly supported by ADM SVM. However, we can use a technique similar to Ether [10] for this task. We modify the index in SYSENTER_CS_MSR to point to some unmapped segment, storing its original value in a safe place. Each time sysenter is called, the CPU will transfer control to this segment and create a #GP fault. MAVMM intercepts this fault to get system call number and other arguments, then passes control back to the guest using original SYSENTER_CS_MSR value, as if no interception has occurred.

[3]sysenter is a recent addition and is only supported in 2.6 series Linux kernels

2) Network & File Access: In Linux, all network accesses are carried out by invoking *sys_socketcall*, which takes two parameters: *func* indicating which network system call to execute, and *args* - an array of pointers to different parameters associated with *func*. When *sys_socketcall* is intercepted, the value of *func* and *args* are located at CPU register `ebx`, and `ecx` accordingly. We can get the IP address as well as the port number of the host to which the guest is communicating with by looking at the *sockaddr_in* structure specified in *args*. For each network access, MAVMM records the IP addresses, port numbers, and data involved.

File accesses can be monitored in a similar fashion through tracking of *sys_read* and *sys_write* system calls. To facilitate analysis, we maintain a mapping from descriptor numbers of opened files to their pathnames. We update this map when intercepting returns of *sys_open* and *sys_close*.

3) Getting Analysis Data: Given that we can use Simnow to bind a virtual serial port in the simulator to a real port on the hosting system, we currently use a serial port for sending out analysis data. Though it has low bandwidth, it serves as a proof-of-concept for our ideas. A similar device hiding and I/O access mechanism could be used for an external USB drive.

4) Selective Analysis: We implemented the MAVMM user control interface using a guest program and VMMCALL instructions. Our program, mavmm-u, running inside the guest makes VMMCALLs to communicate with the hypervisor. We use this program to take fine-grained control of our tracking features, switching between compact mode and full mode, and specifying the names of processes that we want to track. Although mavmm-u is run inside the guest, we can remove the binary file and evidences of its existence before executing the malware.

To track sub-processes, we intercept Linux's *execve* system call, which is the backend of *exec* family of functions. We get the name of the newly created processes from *execve*'s arguments and keep a list of all processes that we want to track. This way we can track execution of sub-processes that are created by the malicious application.

E. Transparent Event Forwarding

The main role of MAVMM is to log actions executed by the guest running on top of it. For most of the time, MAVMM intercepts an event, log it, and then forward it to the guest as if no interception has occurred. Hardware virtualization offers support for forwarding some types of events, such as interrupt and exception. But the range of events that MAVMM needs to intercept is broader than that. For example, MAVMM intercepts IRET instruction and modification of CR3 to track system call return value and process switch accordingly. Since forwading of those two events is not supported, it would be very complex if MAVMM tries to simulate those events by itself. To get around this, we implement a transparent event forwarding

```
(1)
******* GUEST STATE *********
cs:ip = 0x73:0xb7ed3b8c
ss:sp = 0x7b:0xbfdc3c84
ds:bp = 0x7b:0xbfdc3ca0
eax = 0x4, ebx = 0x0, ecx = 0x1, edx = 0xb7f52000
esi = 0xd, edi = 0xb7f52000cpl=0x3
cr0=0x8005003b, cr3=0x0, cr4=0xf1fc000
rflags=0x346, efer=0x0
4 bytes opcode: 0xcd 0x80 0x5b 0x3d
>> write( filename: stdout, size: 0xd, content
written [Hello world!] )
>> syscall return: 0xd
******* GUEST STATE *********
...

(2)
>> unlink( Filename: /etc/passwd~ )
>> link( Old Filename: /etc/passwd, New Filename:
/etc/passwd~ )
>> unlink( Filename: /etc/passwd )
>> link( Old Filename: /etc/ptmp, New Filename:
/etc/passwd )
>> unlink( Filename: /etc/ptmp )
```

Figure 3. MAVMM logs: (1) Trace of a simple program, high level system calls are combined with fine-grained execution trace. (2) Rootkit.Linux.Agent.30.Chsh replaces /etc/passwd through a combination of link and unlink system calls

mechanism using the single stepping TF flag in *rflags* register. When IRET or CR3 modification takes place, MAVMM logs the event, disables interception for that particular event, and then sets TF flag to 1 before returning control to the guest. This way, the guest will receive the event without any alteration. Right after the guest processes this event, a #DB single stepping exception will be raised and control is passed back to MAVMM. This time, MAVMM resets TF and enable interception for the event again. This technique works well for most interceptions except interrupt and exception, because the CPU will reset TF flag at the beginning and restore it at the end of those two events. As a result, #DB exception will not be raised after the first instruction within the interrupt or exception handler, and MAVMM will not be able to reestablish its interception immediately as expected. Fortunately, forwarding of those two events is already supported by hardware virtualization.

IV. Evaluation

We have evaluated MAVMM in three aspects: functionality, detectability, and performance. We executed our experiments inside the AMD Simnow simulator, which simulates a machine with 900Mhz processor and 256MB of RAM. We ran Simnow on a 2.40GHz Intel(R) Core(TM)2 CPU with 2.5GB of RAM, on top ofx86_64 Ubuntu Linux 8.04, kernel version 2.6.24-24.

A. Functionality

1) Fine-grained tracking: MAVMM has the ability to extract very fine-grained information regarding the program under analysis. It can intercept every guest instruction, fetch and display the opcode, CPU registers and other states. This

Detection Technique	VMWare	Virtual PC	Xen PV	Xen HVM	MAVMM
Red Pill (IDT Check)	Detected	Detected	Detected	Not Detected	Not Detected
LDT Check	Detected	Detected	Detected	Not Detected	Not Detected
VMWare I/O Channel	Detected	Not Detected	Not Detected	Not Detected	Not Detected
Virtual PC Special Inst.	Not Detected	Detected	Not Detected	Not Detected	Not Detected
MSW Check	Detected	Not Detected	Not Detected	Not Detected	Not Detected
Xen CPUID Check	Not Detected	Not Detected	Detected	Detected	Not Detected

Table I
EFFECTIVENESS OF VMM DETECTION TECHNIQUES ON VARIOUS HYPERVISORS

is equivalent to the amount of information we can get from a runtime debugger. However, MAVMM offers much better transparency and protection since it operates totally outside the guest. Figure 3-(1) shows a portion of the execution trace when we monitor a simple "Hello world" program. This information can be forwarded to a dissasember for further analysis, or it can be combined with high-level data such as system call traces, to give a clearer picture of the malware as shown in the figure. With fine-grained tracking capability, MAVMM can also be used as a universal unpacker, similar to Ether [10]. The idea is simple: track all memory addresses to which the monitored guest program writes, and raise an alert when it tries to execute dynamically generated code.

2) High-level tracking: To test high-level extraction capability of MAVMM, we monitored the booting process of tty Linux 8.0. During this process, MAVMM intercepted a total of 21953 system calls. Among those system calls 126 are *execve*; they were called to execute binary programs such as *hotplug, chmod, cat, date, stty, mount* and *ifconfig*. Other system calls that we recorded include *read, write, mmap2, ioctl, open* and *close*.

We also reverse-engineered a simple malware to show how our system would work in practice. In order to to this, we downloaded nearly 67000 malware from VXNetlux [2] and used the latest version of ClamAV [1] to remove known samples. Among the remaining ones, we selected a malware named 'Rootkit.Linux.Agent.30.Chsh' due to its small size of roughly 138KB. We enabled system call tracking for this specific process and ran it inside MAVMM with tty Linux 8.0 as our guest OS. We will describe our findings here briefly. The full log can be found at our project page: sourceforge.net/projects/mavmm/files/. By looking at its output messages, we suspect that this rootkit will replace root's shell with a malicious shell so that it gets executed everytime root account logins. But we do not know what the malware actually does or how it achieves its goal. As it turns out, this rootkit tries to change the content of */etc/passwd*. Further analysis reveals that the malware does not modify */etc/passwd* directly; it creates and operates on a temporary copy at */etc/ptmp* instead. This is perhaps to prevent possible errors in the process from destroying the original file, and creating undesired suspicion. After the shell has been changed, the original */etc/passwd* is copied to */etc/passwd*∼ and then get replaced by the modified */etc/ptmp*. Those actions are performed using *link* and *unlink* system calls, as shown in Figure 3-(2).

B. Detectability & Security

We evaluated MAVMM against well-known VMM detection techniques and compared the result with other VMMs such as VMWare, Virtual PC and Xen. The first detection techniques we tested is Red Pill [34]. The idea behind it is very simple: use a sensitive but non-privileged instruction to expose a VMM artifact. Hypervisors such as VMWare and Virtual PC virtualize and relocate the guest interrupt descriptor table (IDT) to a high memory address. Red Pill checks the address stored in IDTR by using the SIDT instruction. If the address stored in IDTR is higher than a certain value, Red Pill concludes that it is running in guest mode, inside a VMM. MAVMM, on the other hand, does not need to modify the IDT table or change value of guest's IDTR, and thus Red Pill is not able to detect MAVMM. Quist and Smith [42] proposed a similar idea, which checks the value of local descriptor table register. We implemented this detection and verified that it is able to detect several other VMMs, but fails to detect MAVMM. We also experimented with some VMM specific detection techniques. For example, VMWare use a special IO port to communicate with the guest. By testing whether this communication is possible, an attacker can detect the presence of VMWare [29]. Virtual PC can also be detected in a similar way [29]. Xen provides modified software MMU architecture, which extends the CPU to improve performance. Therefore Xen, both para-virtualized and hardware-supported versions, can be detected by checking Xen CPUID extensions [16]. It is obvious that MAVMM cannot be detected using these techniques. Finally, we tested MAVMM with the machine state word (MSW) detection technique [41]. This technique can detect fully virtualized VMWare, which cannot be detected by IDT check. The results of our experiments are shown in Table I. As we can see, MAVMM is not susceptible to any of these mechanisms while VMWare, Virtual PC and Xen were susceptible to some of them.

Even though implemented instances of the TLB profiling attack [35] cannot detect MAVMM, we think that the general idea behind it deserves more discussion. In a virtualized environment, both the VMM and the guest compete for the

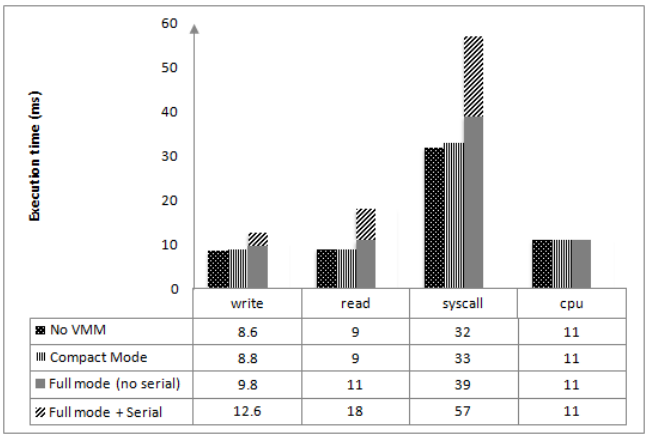
same set of TLB entries. The guest could execute several different types of TLB profiling. One method is to fill all TBL entries, then measure how long it would take to access memory in two different runs: before entering to and after exiting from the hypervisor. If some of the entries filled by the guest get replaced by the VMM, the second run will experience cache misses and take longer to execute [17]. In all scenarios, the guest has to create a #VMEXIT so that hypervisor mode is entered. To achieve higher detection accuracy, this event should not be intercepted by the guest OS, i.e. it is not privileged, otherwise it would be unclear wether the guest OS or the hypervisor caused the timing disparity. Since MAVMM does not support multiple guest VM instances and virtual device emulation, it has no need to virtualize non-privilege instructions such as SIDT or SLDT. It should be much more difficult, if not impossible, to detect MAVMM using TLB profiling attack. Furthermore, MAVMM occupies a much smaller code region than general purpose VMMs, and therefore it will overwrite fewer TLB entries, making this attack more error-prone.

Nevertheless, complete undetectability is like a panacea. We speculate that a carefully implemented and specially targeted external timing attack [14] can be used to detect all VMMs, including our hypervisor. However, such attacks are very complex and expensive. It requires root privilege, a huge amount of CPU cycles, an external timing source and some prior knowledge about the target system. This goes directly against common malware's incentive to be stealthy and remains undetected for a longer period of time and therefore is unlikely to get implemented in practice. Additionally, the growing usage of VMMs in general purpose operating systems, such as the upcoming version of Windows Server, will make VMM detection irrelevant. This will force attackers to instead focus on the more difficult problem of *analyzer detection*.

The size of trusted computing base is an important factor to consider when evaluating a system's security. Simplicity makes it easier to avoid bugs, and to formally verify desired properties of the system. Our current implementation consists of 182 lines of assembly and 3913 lines of C code for the hypervisor, and 75 lines of C code for the user control interface. After compiled, the MAVMM hypervisor is only 124KB. Our code base is 3 to 4 order of magnitude smaller than commodity VMMs such as Xen or VMWare Workstation, which contain a host OS with millions of lines of code inside their TCB.

C. Performance Overhead

Even though performance overhead is not our main concern, we want make sure that it can be kept reasonable. We evaluated the performance of MAVMM by measuring execution time of different types of programs inside (in both compact mode and full mode), and outside our hypervisor. We ran each program five times and show the average of all



	write	read	syscall	cpu
No VMM	8.6	9	32	11
Compact Mode	8.8	9	33	11
Full mode (no serial)	9.8	11	39	11
Full mode + Serial	12.6	18	57	11

Figure 4. MAVMM performance overhead. Each group has three bars. First bar is execution time when MAVMM is disabled, second bar with MAVMM in compact mode, last bar with MAVMM in full mode, the upper portion of this bar is additional delay incurred by serial port communications

runs in Figure 4. First, we ran two I/O intensive programs, one reads (*read*) and the other writes (*write*) one million bytes to the disk. Then we executed another program that makes 1000 getpid() system calls and print out the result to the screen (*syscall*). Lastly, we ran a CPU intensive program that execute one million *add* instructions (*cpu*). As we expected, the last program does not experience any performance penalty since all executed operations are non-privilege. The result shows that MAVMM, when running in compact mode, induces very negligible overhead: 0% for *read* and *cpu*, 2.3% for *write* and 3.1% for *syscall*. However, the extra cost in full mode is much more significant: 46.5%, 100%, and 78.1% for *write, read* and *syscall* respectively. Further investigation reveals the main source of overhead is the serial port communication. Our current implementation simply dumps all logged data to COM1. This step takes up 70% to 77.8% of the three additional delays mentioned above. This expense will be reduced significantly when we switch to usb logging and batch data dumps rather than writing them as they happen. Since we have not tried to optimize our code our code heavily, we believe that aggressive optimization, such as using local caching, will achieve even better performance.

V. Related Works

A significant motivation for our project is prior works on malware analysis in non-virtualized environment, including in-guest debugger [48] and dissasembler [36]. Those techniques, however, can be avoided through various number of methods such as packing/encryption, code obfuscation [43], and debugger detection [19]. More advanced systems include OS based platforms such as Saffron [43], and emulator based analyzers such as Renovo [26]. Saffron uses dynamic instrumentation and a newly developed page fault assisted debugger, while Renovo and BitBlaze [44] ultilize whole-

system emulation. However, they only provide a way to debug / unpack malware whereas MAVMM offers a more complete analysis platform. Moreover, non-virtualized analyzers are very likely to create detectable side effects, especially when they operate under the the assumption that the guest OS can be compromised. Our goals of minimal detectability and no trust on the guest, including guest OS, cannot be accomplished in this environment.

Virtualization offers a strong protection through isolation, and the ability to save and rollback guest state to aid live debugging. VM introspection, the process of examining a process inside a virtual machine from its VMM, was introduced by Garfinkle and Rosenblum [18]. While other works have leveraged this idea for security purposes, such as process tracking [24], intrusion detection [28, 46], malware detection [22], and honeypots [30, 23, 32], our work focuses on *harware-supported* introspection for malware analysis.

Because virtual machines have been used commonly by malware analyzers, virtualization detection techniques have become a part of modern malware. The techniques that malware program use range from a simple IDT based detection [34] to complicated TLB sizing or timing attacks [17, 13]. These results show that any software virtualization platform will introduce some detectable changes to the guest system. We, therefore, utilize the hardware virtualization support to achieve our goal of minimal detectability. Since it is known to be harder to detect hardware virtualization, malware is unlikely to go to great length to detect and avoid hardware virtualization platform if by doing so exposes itself to malware detectors.

Several researches are utilizing hardware virtualization. KVM [20] uses kernel modules to create a hypervisor on top of Linux, but it is based on QEMU's I/O model which is known to be detectable [33, 13]. A recent work by Dinaburg et al, Ether [10], is perhaps the project most closely related to ours. Ether make use of Xen HVM and its support for Intel VT hardware virtualization technology for malware analysis. Intel VT, however, does not support nested paging and DMA protection. This is the reason why we decided to use AMD SVM instead of Intel VT. The usage of Xen makes it much easier to develop Ether, since the analyzer does not have to worry about boot-strapping itself and the guest OS, protecting its integrity, or retrieving analysis data, etc... But this benefit comes at the cost of having a huge TCB. Ether's trusted computing base includes Xen and an additional domain0 OS with many unnecessary functionalities. As we have argued throughout this paper, general purpose VMMs are not appropriate for malware analysis. We on the other hand, use a lightweight and customized VMM which is specially designed for this purpose.

Another interesting line of works uses a thin layer of hypervisor to enforce guest security policies [40], to help reducing the TCB size of guest applications [15], or to implement a low level rootkit and hide its malicious behaviours [35]. Our work also focuses on separating functionalities and keeping TCB at minimum, but we target a different application which presents a different set of technical challenges.

VI. Conclusion

In this paper, we proposed MAVMM, a lightweight VMM designed specially for malware analysis. MAVMM does not implement unnecessary virtualization features commonly found in general purpose hypervisors. Hardware virtualization support offers MAVMM simplicity, security and transparency. We proved that our system can extract useful information, and that it is not susceptible to known virtualization detection techniques. Thus, it can achieve higher accuracy than current state-of-the-art malware analysis platforms.

Another important goal that we started with was to provide the research community with a simple and easy-to-enhance hardware-supported virtualization framework. This framework could be useful for prototyping new functionality below OS level. Such services include OS debugging, security auditing, logging, and replaying, etc. By being simple (around 4000 lines of code) and well documented (even larger amount of comments), MAVMM makes it easy for other researchers to add new functions to it, or modify it to serve their purposes. Our implementation of MAVMM and updates on our project can be found at mavmm.sourceforge.net

Acknowledgment

We would like to acknowledge Professor Carl A. Gunter for his thoughtful suggestions during the early stage of this project. This work was funded by a grant from the Internet Services Research Center (ISRC) of Microsoft Research, by NSF grant CT-0716768, and by AFOSR MURI grant FA9550-09-01-0539. Anh Nguyen was funded in part by a grant from the Vietnam Education Foundation (VEF). The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of the sponsors.

References

[1] Clam antivirus. http://www.clamav.net/.

[2] Vx heavens. http://vx.netlux.org/.

[3] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.14 edition, Sep 2007.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, 2003.

[5] V. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation, 1993.

[6] D. P. Bovet and M. Cesati. *Understanding the linux kernel*. Number ISBN : 0-596-00213-0. O'Reilly, décembre 2003.

[7] E. Chickowski. Webroot: 40 percent of companies report disruptions due to malware. *SC Magazine*, Mar 2007.

[8] G. Combs. Wireshark 1.0.6: A Network Protocol Analyzer for Windows and Unix. http://www.wireshark.org/.

[9] W. Davis. Adware Firms Up The Ante On Anti-Spyware. Online Media Daily, Mar 2005.

[10] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, 2008.

[11] C. Economics. 2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnests and Other Malicious Code. *Tech. Rep.*, Jun 2007.

[12] F-Secure. Agobot virus description. www.f-secure.com/v-descs/agobot.shtml.

[13] P. Ferrie. Attacks on virtual machine emulators, Jan 2007.

[14] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. V. Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating System Review Special Edition on Computer Forensics*, 42(3):83–92, Apr 2008.

[15] M. Franz. Information-flow aware virtual machines: Foundations for trustworthy computing. *Conference For Homeland Security, Cybersecurity Applications and Technology*, 0:91–96, 2009.

[16] K. Fraser. x86: Update xen-detect utility to scan for Xen signature in CPUID space, Dec 2008. xen-unstable mailing list.

[17] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Usenix Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *ISOC Network and Distributed Systems Security Symposium*, Feb 2003.

[19] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. *IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop*, 2005.

[20] Q. Inc. KVM - Kernel-based Virtualiztion Machine.

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*, Mar 2009.

[22] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *ACM conference on Computer and communications security*, 2007.

[23] X. Jiang, D. Xu, H. Wang, and E. Spafford. Virtual playgrounds for worm behavior investigation. In *International Symposium on Recent Advances in Intrusion Detection*, 2005.

[24] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference*, 2006.

[25] K. Kaneda. Tiny Virtual Machine Monitor. http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/.

[26] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *ACM Workshop on Recurring Malcode (WORM'7)*, October 2007.

[27] S. King and P. Chen. Subvirt: Implementing malware with virtual machines. *IEEE Symposium on Security and Privacy*, May 2006.

[28] K. Kourai and S. Chiba. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *ACM/USENIX international conference on Virtual Execution Environments*, 2005.

[29] T. Liston and E. Skoudis. On the cutting edge: Thwarting virtual machine detection, Jul 2006.

[30] Y. min Wang, Y. min Wang, D. Beck, D. Beck, X. Jiang, X. Jiang, R. Roussev, and R. Roussev. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS*, 2006.

[31] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[32] N. Provos and T. Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, 2007.

[33] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *ISC*, 2007.

[34] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.invisiblethings.org/papers/redpill.html.

[35] J. Rutkowska and A. Tereshkin. IsGameOver() Anyone?, May 2007. http://bluepillproject.org/stuff/IsGameOver.ppt.

[36] D. SA/NV. IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro/.

[37] Secunia. Vulnerability Report: VMware ESX Server 3.x. http://secunia.com/advisories/product/10757/.

[38] Secunia. Vulnerability Report: VMware Workstation 6.x. http://secunia.com/advisories/product/10757/.

[39] Secunia. Vulnerability Report: Xen 3.x. http://secunia.com/advisories/product/15863.

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07*, 2007.

[41] V. Smith and D. Quist. Further Down the VM Spiral, Aug 2006. www.offensivecomputing.net/dc14/furthur_down_the_vm_spiral.pdf.

[42] V. Smith and D. Quist. Hacking Malware: Offense is the new Defense. *Defcon*, 14, Aug 2006.

[43] V. Smith and D. Quist. Covert Debugging: Circumventing Software Armoring. *Blackhat Las Vegas*, Aug 2007.

[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, Dec 2008.

[45] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference*, 2001.

[46] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39(5):148–162, 2005.

[47] R. Wojtczuk. Adventures with a certain Xen vulnerability(in the PVFB backend), Oct 2008.

[48] O. Yuschuk. Ollydbg. http://www.ollydbg.de/.

[49] B. Zdrnja. E-cards dońt like virtual environments, Jul 2007. http://isc.sans.org/diary.html?storyid=3190.

[50] B. Zdrnja. More tricks from Conficker and VM detection, Feb 2009. http://isc.sans.org/diary.html?storyid=5842.

[51] L. Zeltser. Using VMware for Malware Analysis. *SearchSecurity.com*, May 2007.