Stealthy Malware Detection and Monitoring Through VMM-Based "Out of the Box" Semantic View Reconstruction

XUXIAN JIANG
North Carolina State University
XINYUAN WANG
George Mason University
and
DONGYAN XU
Purdue University

Authors' addresses: Xuxian Jiang, Department of Computer Science, North Carolina State University, 890 Oval Drive, Raleigh, NC 27695; email: jiang@cs.ncsu.edu. Xinyuan Wang, Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030; email: xwangc@gmu.edu. Dongyan Xu, Department of Computer Science and CERIAS, Purdue University, 305 N. University Street, West Lafayette, IN 47907; email: dxu@cs.purdue.edu

An alarming trend in recent malware incidents is that they are armed with stealthy techniques to detect, evade, and subvert malware detection facilities of the victim. On the defensive side, a fundamental limitation of traditional host-based anti-malware systems is that they run inside the very hosts they are protecting ("in the box"), making them vulnerable to counter-detection and subversion by malware. To address this limitation, recent solutions based on virtual machine (VM) technologies advocate placing the malware detection facilities outside of the protected VM ("out of the box"). However, they gain tamper resistance at the cost of losing the internal semantic view of the host, which is enjoyed by "in the box" approaches. This poses a technical challenge known as the *semantic gap*.

In this paper, we present the design, implementation, and evaluation of *VMwatcher* – an "out of the box" approach that overcomes the semantic gap challenge. A new technique called *guest view casting* is developed to reconstruct internal semantic views (e.g., files, processes, and kernel modules) of a VM non-intrusively from the outside. More specifically, the new technique casts semantic definitions of guest OS data structures and functions on virtual machine monitor (VMM)-level VM states, so that the semantic view can be reconstructed. Furthermore, we extend guest view casting to reconstruct details of system call events (e.g., the process that makes the system call as well as the system call number, parameters, and return value) in the VM, enriching the semantic view. With the semantic gap effectively narrowed, we identify three unique malware detection and monitoring capabilities: (1) *view comparison-based malware detection* and its demonstration in rootkit detection; (2) *"out of the box" deployment of off-the-shelf anti-malware software* with improved detection accuracy and tamper-resistance; and (3) *non-intrusive system call monitoring* for malware and intrusion behavior observation. We have implemented a proof-of-concept VMwatcher prototype on a number of VMM platforms. Our evaluation experiments with real-world malware, including elusive kernel-level rootkits, demonstrate VMwatcher's practicality and effectiveness.

Categories and Subject Descriptors: D.4.6 [**Operating System**]: Security and protection – Invasive software; K.6.5 [**Management of Computing and Information Systems**]: Security and protection

General Terms: Security

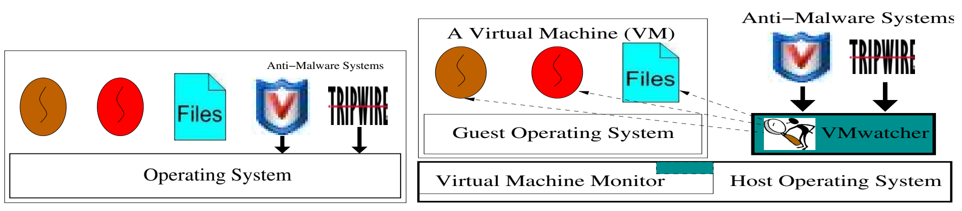
Additional Key Words and Phrases: Malware Detection, Rootkits, Virtual Machines

1. INTRODUCTION

Internet malware (e.g., rootkits, worms and bots) is getting increasingly stealthy and elusive: they try to hide their presence from detection facilities and even detect and subvert any existing anti-malware software in the compromised system. For example, a detailed analysis of an Agobot variant [Agobot 2004] has revealed that the malware contains malicious logic to detect and remove more than 105 anti-virus processes in the victim machine.

The threat above is partly attributed to a fundamental limitation on the defensive side: Most host-based anti-malware systems are installed and executed inside the very hosts that they are monitoring and protecting (Figure 1(a)). Although such "in the box" deployment provides an anti-malware system with a native, semantic-rich view of the host, it in the meantime makes the anti-malware system visible, tangible, and potentially subvertable to advanced malware residing in the host.

To address this problem, there have recently been a number of solutions [Dunlap et al. 2002; Garfinkel and Rosenblum 2003; Joshi et al. 2005] that advocate placing the intrusion detection facilities outside of the (virtual) machine being monitored. Based on virtual machine technologies [Barham et al. 2003; Dike 2002], such an "out of the box" approach significantly improves the tamper-resistance of intrusion detection facilities. A virtual ma-



(a) Traditional “in the box” approach (b) VMwatcher approach

Fig. 1. Malware detection in traditional “in the box” approach and in VMwatcher approach

chine (VM) achieves strong isolation and confines processes running inside the VM such that, even if they are compromised by malware, it will be hard, if not impossible, to compromise systems outside of the VM.

However, a dilemma exists in switching from the “in the box” approach to the “out of the box” approach: It is well-known that there exists a “semantic gap” [Chen and Noble 2001] between the view of the VM from the outside and the view from the inside – the latter being seen by the traditional, “in the box” anti-malware systems. For example, instead of seeing semantic-level objects such as processes, files, and kernel modules, we only see memory pages, registers, and disk blocks from outside the VM, making “out of the box” malware detection difficult. In other words, the “out of the box” approach gains tamper resistance at the cost of losing the internal semantic view of the host enjoyed by the “in the box” approaches.

The above dilemma motivates us to explore the possibility of gaining the advantages of both camps, namely enabling tamper-resistant malware detection *without losing the semantic view*. In this paper, we present the design, implementation, and evaluation of VMwatcher – a VMM-based, “out of the box” approach that overcomes the semantic gap challenge. More specifically, VMwatcher instantiates the general virtual machine introspection (VMI) [Garfinkel and Rosenblum 2003] methodology in a non-intrusive manner, so that it can inspect the low-level VM states and events without perturbing the VM’s execution. A new technique called *guest view casting* is developed to systematically reconstruct the VM’s internal semantic view (e.g., files, directories, processes, and kernel-level modules) for out-of-the-box malware detection. Furthermore, we extend guest view casting to reconstruct details of system call events in the VM (e.g., the calling process as well as the system call number, parameters, and return value). The new technique is based on the key observation that the guest OS of a VM provides all necessary semantic definitions of guess OS data structures, functions, and system calls to construct the VM’s semantic view. As such, we can cast these definitions on the VMM-level observations and externally derive the semantic view of the target VM (Figure 1(b)).

VMwatcher enables new malware detection and monitoring capabilities that are previously difficult or impossible to achieve. In this paper, we identify and demonstrate three such capabilities: (1) *view comparison-based stealthy malware detection*, which involves comparing a VM’s semantic views obtained from both inside and outside for possible discrepancy detection; (2) *out-of-the-box execution of unmodified, off-the-shelf anti-malware software* with improved detection accuracy. This is an extreme test to VMwatcher’s semantic gap-narrowing technique and, interestingly, it further enables *cross-platform* malware scanning where anti-malware software developed for one platform can be readily used for another platform; (3) *non-intrusive system call monitoring* in a production or honeypot VM, which elevates the tamper-resistance of malware behavior observation and experimentation.

We have implemented a VMwatcher prototype on a number of VMM platforms and evaluated it with a collection of real-world malware instances (e.g., kernel and user level rootkits). Experiments with these elusive rootkits demonstrate VMwatcher's unique capability of view comparison-based malware detection. The VMwatcher prototype also supports out-of-the-box deployment of a variety of off-the-shelf anti-malware software such as Symantec AntiVirus and Microsoft Windows Defender.

The rest of this paper is organized as follows: Section 2 presents the design of VMwatcher, followed by the implementation details in Section 3. We present evaluation results in Section 4 and discuss possible limitations in Section 5. Section 6 discusses related work and Section 7 concludes this paper.

2. VMWATCHER OVERVIEW

2.1 Design Goals and Assumption

Figure 1 illustrates the key difference between the traditional "in the box" approach and the VMwatcher approach for malware detection. VMwatcher achieves stronger tamper-resistance by moving malware monitoring facilities out of the VM being monitored. VMwatcher is based on two key enabling techniques: (1) non-intrusive VM introspection for the procurement of low-level (VMM-level) VM states and system call events, without deploying on any facility inside the VM (Section 2.2.1) and (2) guest view casting for external reconstruction of VM internal semantic view (Section 2.2.2). VMwatcher has the following three design goals:

First, VMwatcher should not perturb the system state of the target VM. This will prevent VMwatcher from affecting the normal execution of the VM and causing adverse side effects (e.g., system inconsistency [Joshi et al. 2005]) in the VM. This goal is realized by our technique for non-intrusive inspection and analysis of low-level VM observations. Non-intrusiveness also makes it hard for internal malicious processes to infer (external) VMwatcher activities.

Second, VMwatcher should significantly narrow the semantic gap, such that the same malware detection system that runs inside the VM can also run outside of the VM. As to be shown, this goal is critical to the new malware detection capabilities. The goal is realized by our guest view casting technique for external reconstruction of VM semantic view. Based on the reconstructed view, anti-malware systems can perform file or memory scanning operations as if they were inside the VM [1].

Third, VMwatcher should be generic and applicable to a number of existing VMMs. Currently there exist two mainstream virtualization approaches: full virtualization and para-virtualization. Full virtualization (as in VMware [VMware 2008] and QEMU [Bellard 2005]) transparently supports legacy OSes without modifying the guest OS code; while para-virtualization (as in Xen [Barham et al. 2003] and User-Mode Linux [Dike 2002])

[1]We need to point out that some hooking-based features of anti-malware systems are hard to support by VM introspection. Certain high-level events (e.g., Windows API calls or hooks), which are of interest to some anti-virus software, may not be captured from low-level VMM observations.

is less transparent as it needs to modify the guest OS source code. VMwatcher aims at supporting VMMs in both categories.

We also note that different VMMs choose to implement VMs at different levels, imposing varying complexity on VMwatcher. More specifically, the lower the virtualization level, the wider the semantic gap it will create and, consequently, the greater the challenge for VMwatcher to bridge the semantic gap. For example, because of its system call level virtualization, User-Mode Linux (UML) preserves much of the semantic information (e.g., processes) and thus leads to a much narrower semantic gap than VMware, Xen, and QEMU.

Assumption on trusted VMM In this paper, we assume a trusted VMM that achieves VM isolation: A malware instance may compromise arbitrary entity and facility inside the VM – including the guest OS kernel itself. However, it cannot break out of the VM and corrupt the underlying VMM. This assumption is based on the observation that the code base of a VMM is much smaller and more stable than the legacy OS code. Further, the VMM provides a more limited interface (which can be further hardened and validated) to untrusted VMs in the form of virtualized underlying physical resources. We note that this assumption is consistent with that of many other VM-based security research efforts [Dunlap et al. 2002; Garfinkel et al. 2003; Garfinkel and Rosenblum 2003; Joshi et al. 2005; Koju et al. 2005]. We will discuss possible attacks (e.g., VM fingerprinting) in Section 5.

2.2 Enabling Techniques

2.2.1 *Non-Intrusive Virtual Machine Introspection.* VMwatcher follows the VM introspection methodology to capture low-level VM states and events externally. For open-source VMMs such as Xen, QEMU, and UML, we develop VM introspection extensions to obtain full VM state which includes the VM's registers, memory, and disk and to capture system calls made by processes in the VM. To achieve non-intrusiveness, we follow the principle of passive, read-only observation without inflicting any influence on the VM – this is important as such an influence would lead to undesirable consequences such as inconsistency in the VM's system state or perturbation in the VM's execution.

For close-source VMMs, we only have limited access to VMM-level observations. For example, with Microsoft Virtual PC, we are not able to read VM registers (e.g., the control register CR3) or monitor virtual interrupts. Without a VMM's source code, VMwatcher has to rely on whatever low-level VM state abstraction exposed by the VMM. Details of our non-intrusive VM introspection technique will be presented in Section 3.1.

2.2.2 *Guest View Casting.* Given the VMM-level observations of a running VM, our second technique, guest view casting, will externally reconstruct the internal semantic view of the VM. We observe that the guest OS data structure definitions (e.g., files and directories) and function semantics (e.g., semantics of file system drivers) can be used as "templates" to interpret low-level VM states. As such, we can cast the guest data structure and function definitions on the VMM-level VM observations to derive the VM's semantic view. For example, given a "live" virtual disk of a running VM, the guest functions such as guest device drivers and related file system drivers allow us to reconstruct semantic information such as files and directories from the "raw" bits and bytes on the virtual disk. Similarly, by casting guest memory data structures (e.g., process control blocks) and functions to the physical memory pages allocated to a VM by the VMM, we can identify each individidual running process with its attributes such as process id and name, and derive semantic information about each loaded kernel module inside the VM.

Guest view casting further performs high-fidelity restoration of semantic objects, so that the restored objects can be presented to an anti-malware system in exactly the same way as inside the VM. For example, Tripwire [Kim and Spafford 1994] assumes a standard UNIX-like file system layout and calculates the checksums of files and directories to identify possible changes; McAfee VirusScan examines file directories and attempts to spot any existing malware in these directories. As such, guest view casting needs to further "package" the objects (e.g., files and directories) in the reconstructed semantic view and seamlessly present these objects to the anti-malware system in their native, manipulable form.

2.2.3 *Guest System Call Reconstruction.* Guest view casting reconstructs a VM's semantic state. For malware monitoring and detection, it is also desirable to capture and interpret the system call events that occur inside the VM. To this end, we extend the guest view casting technique to enable guest system call reconstruction. More specifically, processes (including the malicious ones) inside the VM make system calls by executing system call instructions (e.g., *sysenter/sysexit*). Such an instruction will be captured by the VMM. Our virtual machine introspection technique will then be invoked to further acquire low-level context information relevant to the system call (e.g., register values and memory contents). This context information will be interpreted by our extended guest view casting technique, using system call semantics as the casting "templates". Guest system call reconstruction generates detailed system call information, including the process making the system call and the system call number, parameters and return value. The reconstruction of guest system calls is performed in real-time and from outside the VM, which improves the tamper-resistance of existing system call-based monitoring and detection systems.

2.3 New Malware Detection and Monitoring Capabilities

VMwatcher enables a number of useful malware detection and monitoring capabilities. The first capability is view comparison-based detection of elusive malware. We have seen an increasing number of elusive malware instances that hide themselves (including the related files and processes) by subverting anti-malware processes running inside the system. For view comparison, we corroborate an internal view (generated from inside the VM) with an external view (generated from outside the VM by VMwatcher) of the same objects of interest and detect the existence of hidden malware based on any view discrepancy exhibited. We note that view comparison can be performed either on the full semantic views of a VM, or on more focused, customized views (e.g., a list of files/processes satisfying a certain condition) generated by a malware detection function. As an example, running the *ls* command inside a Linux VM can provide an internal view of those files under the current directory. With VMwatcher, we can run the same *ls* command outside of the VM and obtain an external view of the files under the same directory. Any difference between the two *ls* results will immediately lead to the detection of hidden files.

View comparison is not limited to a VM's persistent states such as disk files. It can also be performed on the VM's volatile states such as running processes, loaded kernel modules, or even current statistics of a NIC device. We find this capability highly valuable, especially when detecting advanced kernel-level rootkits that hide running processes or kernel modules (Section 4.1). We point out that view comparison would be infeasible without VMwatcher: If separated by a semantic gap, the internal and external views of a VM would not be directly comparable.

The second capability is “out of the box” execution of off-the-shelf anti-malware systems, which improves the detection accuracy as well as tamper-resistance of these systems. Moreover, since the guest OS of a VM may be different from the host OS, it is possible to perform cross-platform malware detection, where anti-malware software developed for one platform (e.g., Windows) can be readily used for another platform (e.g., Linux). We will present one such experiment in Section 4.2.

The third capability is non-intrusive system call monitoring for malware behavior observation. With VMwatcher's guest system call reconstruction technique, it is possible to monitor system calls made by any process inside a VM, without installing any logging module in the VM or modifying the guest OS. This capability has direct applications in a number of scenarios, such as system call-based anomaly detection [Provos 2003], forensic analysis [King and Chen 2003], and malware experimentation [Jiang and Xu 2004; Jiang et al. 2005].

3. IMPLEMENTATION

We have implemented a proof-of-concept VMwatcher prototype on top of four existing VMMs: VMware, QEMU, Xen, and UML[2]. The prototype is able to reconstruct semantic views of a variety of VMs, including Windows 2000/XP, Red Hat Linux 7.2/8.0/9.0, and Fedora Core 1/2/3/4. In the following, we describe VMwatcher implementation in detail.

3.1 VMM-Level State and Event Procurement

As mentioned in Section 2.1, VMwatcher is designed to be generically applicable to various VMMs. Table I lists the VMM-level VM state and event observation offered by the four VMMs. The open-source VMMs – QEMU, Xen, and UML – allow full access to low-level VM states and events. The close-source VMware typically exposes only the raw disk blocks and raw memory pages allocated to a VM. We recently obtained the source code of VMware Workstation 6.0 through the VMware Academic Program. As a result, our current VMwatcher prototype is able to access full VM states and events.

VMM-level observation	Full virtualization		Para-virtualization	
	VMware	QEMU	Xen	UML
Raw VM disk image	√	√	√	√
Raw VM memory image	√	√	√	√
Other VM hardware states (e.g., registers)	√	√	√	√
VM-related low-level events (e.g., interrupts)	√	√	√	√

Table I. VMM-level observation of VM states and events

For the procurement of the VM's raw disk and memory states, we need to access a VM's raw disk and memory while they are being modified. To ensure state consistency, a VMM usually grants exclusive access (e.g., with a *write* lock) to virtualized resources (e.g., memory or disk) to a VM. As a result, it could prevent any external process from accessing

[2]In our current prototype guest system call reconstruction is supported by VMware and QEMU because of their convenient system call instruction interception.

them. More specifically, the file lock in Windows imposed by VMware is *mandatory*, which means that any external process such as VMwatcher is not able to read the locked file. There are two possible solutions: One is to follow the same approach taken by current system backup software, which utilizes the Volume Shadow Copy Service (more details in [Microsoft 2003]) of Windows to access the locked files. In other words, we can create a shadow copy of the locked file and instruct VMwatcher to access the shadow copy for VM state procurement. The other approach is to develop a device driver that essentially subverts the host Windows kernel and allows VMwatcher to read the locked file directly through the device driver. Our prototype takes the first approach, which follows the non-intrusive principle as it will not modify the locked file. On UNIX platforms, the file lock is *advisory* by default, which means that we can ignore the lock and just read the locked file.

The above strategy resolves the "read-write" conflict between running VMs and VMwatcher when both are simultaneously accessing the same disk file in the host domain. Note that for a running VM, a file emulating its virtual disk means a root file system or a hard disk partition; while for VMwatcher it is considered the externally observable VM disk state. We also note that VMware, QEMU with KQEMU [Bellard 2006] support, and UML generate a temporary memory file to emulate the allocated raw physical memory for a VM, which allows for external simultaneous access by VMwatcher. However, Xen and QEMU without KQEMU support do not create such memory file. As such, we need to extend them to export a VM's physical memory pages. In our prototype, VMwatcher takes advantage of the *libxc* library [Xen 2004] to access the memory of a Xen-based VM (or DomU) by mapping its physical memory to its address space with the *xc_map_foreign_range()* API and then reading the content through the mapped memory. Similarly, we build our own library for QEMU, which essentially allows for external VMwatcher access to the allocated physical memory pages for a QEMU-based VM.

For the capture of a VM's low-level events, we first leverage a VMM's capability of intercepting system call instructions. Such capability is readily available in a number of VMMs such as VMware and QEMU. Upon the capture of a system call instruction, VMwatcher will be invoked to collect relevant low-level context information about this system call, such as the values of registers (CR3, ESP, EAX etc.) and certain memory contents in the virtual address space of the process that makes the system call. In particular, the procurement of the memory contents is guided by certain register values, which supply the virtual addresses of the corresponding memory contents. With the low-level register and memory states, we extend the guest view casting technique by casting system call semantics to the low-level context information. The extended technique will reconstruct detailed semantic information about the system call event (Section 3.2.3).

3.2 Semantic View Reconstruction

Based on raw VM disk and memory states, VMwatcher uses the guest view casting technique to extract high-level semantic information (e.g., files and processes) and then present them seamlessly to anti-malware software. In the following, we describe our casting methods for disk and memory state reconstruction and for guest system call reconstruction.

3.2.1 *Disk State Reconstruction.* It is straightforward to reconstruct the semantic view from the raw virtual disk blocks of a VM, if we understand how files and directories are organized in the virtual disk. Particularly, our method casts the corresponding device drivers and file system drivers of the guest OS for disk semantic view reconstruction. For Linux, the casting is convenient as the device drivers and file system drivers are likely part of the open-source Linux kernel. However, this is not the case for Windows. The reason is that the Windows kernel does not have the corresponding file system drivers for the Linux root file systems. For the VMwatcher prototype, we have written Windows device drivers to interpret Linux file systems (*ext2/ext3* root file systems).

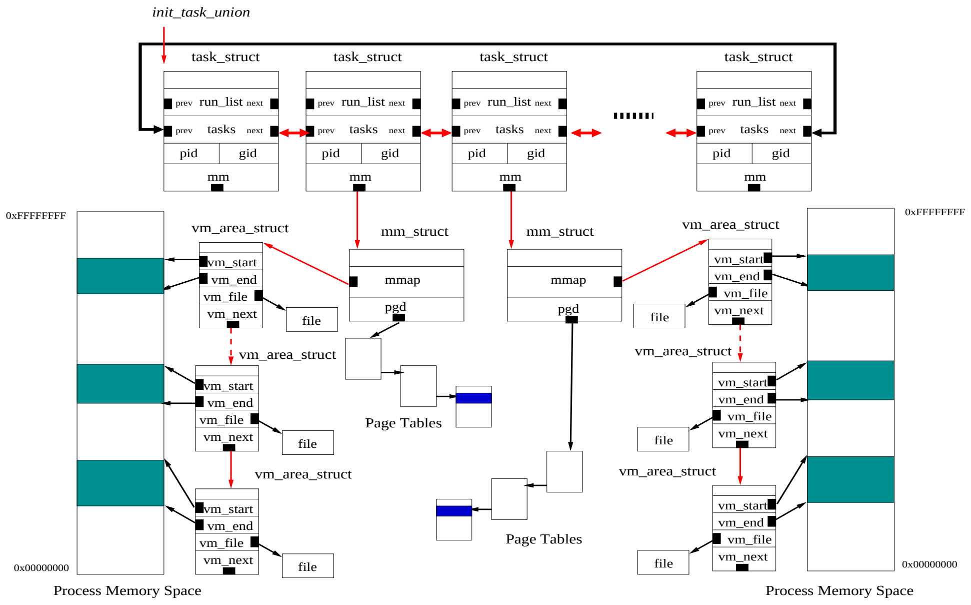


Fig. 2. Guest view casting for volatile VM memory state (Linux)

3.2.2 *Memory State Reconstruction.* It is a more challenging task to reconstruct the semantic view of the volatile VM memory. The challenge is that it requires accurate casting of guest memory data structures and functions to understand how the physical memory pages are utilized. Note that the casted guest memory data structures and functions are specific to a VM kernel.

For ease of presentation, we focus our discussion on Linux for the current 32-bit architecture (which implies an addressable memory range [0, 4G-1]). In Linux, the 4G memory space of a process is split between user space (the bottom 3GB memory) and kernel space (the top 1GB memory) and the Linux kernel is mapped into every user-level process starting at virtual address $0xC0000000$. Based on the physical memory layout, the first Linux kernel page (with virtual address $0xC0000000$) is located in the first physical memory page (with physical address $0x00000000$). This provides the starting point for our guest view casting method: If we can access the memory file containing the raw memory of a running VM, offset 0 in the memory file will correspond to the current memory address $0xC0000000$ inside the VM. Next, we utilize the exported symbol information [3] and apply

[3]For some commercial OSes, the locations of these symbols may not be provided. VMwatcher will perform a full scan of the raw memory and identify the symbols by looking for certain “signatures” [bugcheck 2006] that

guest view casting to identify and reconstruct guest data structures of interest. Figure 2 shows how guest view casting is applied to reconstruct the volatile kernel memory state of a Linux VM. More specifically, every process in Linux is represented by a process control block (defined as *task_struct*) and all running processes are linked by a doubly linked list. The head of this list is kept in a structure called *init_task_union*, which is exported and can be identified by querying the *System.map* file. Following this pointer, we can further parse the raw memory image and traverse the doubly linked list to reconstruct detailed semantic information about each running process (e.g., its page table and memory layout in the *mm_struct* data structure).

From the same memory image, we can also cast and reconstruct a number of other important kernel data structures (e.g., the system call table, the interrupt descriptor table, and the kernel module list) and identify the areas that contain core kernel instructions or instructions in the loadable kernel modules. It is worth mentioning that a user-level memory address ($< 3G$) is usually a virtual memory address of a process running in the VM. Since VMwatcher is running outside of the VM, it needs to translate the virtual memory address into the corresponding physical memory address, which can then be used to access the VMM-level memory state.

We note that existing hardware has the capability of automating the traversal of page table for address translation. However, it implicitly assumes that the virtual address being translated belongs to the current process whose page table base is in CR3. For the virtual address of an *arbitrary* process, VMwatcher will have to externally identify and walk through the page table of that process to obtain the corresponding physical address and read its content. The code for this operation is shown below in function *vmwatcher_vir_mem_read32*, where *addr* is the virtual address to be accessed; *task* points to the process control block (assuming the *task_struct* data structure in Figure 2) of the process of interest; *pde* and *pte* refer to a page directory entry and a page table entry associated with the process, respectively; and *vmwatcher_phy_mem_read32* reads the memory content at the physical address from the VMM-level memory state procured by VMwatcher.

```
unsigned int vmwatcher_vir_mem_read32(task, addr) {

        /* Step 1: obtain the page directory entry */
        pde_addr = task->mm->pgd + (addr >>20) &~3;
        pde      = vmwatcher_phy_mem_read32(pde_addr);

        /* Step 2: obtain the page table entry */
        if ( !(pde & PG_PRESENT) ) return -1;
        pte_addr = pde&~0xfff + (addr >> 10) & 0xffc;
        pte = vmwatcher_phy_mem_read32(pte_addr);

        /* Step 3: obtain the physical address */
        if ( !(pte & PG_PRESENT) ) return -1;
        phy_addr = pte&~0xfff + addr&0xfff;
        return vmwatcher_phy_mem_read32(phy_addr);
}
```

Although the description above is in the context of Linux, guest view casting reflects a generic, systematic methodology that can be applied to various VMM platforms and OSes. During the prototype implementation, we have evaluated how different OSes, service patches, and system configurations impact the casting of VM states and events. For

are unique to kernel-level data structures of interest. For example, we use $0x03001b0000000000$ to identify potential process instances in the Windows XP raw memory file.

example, OSes may have different memory layout (e.g., Windows has a 2G/2G memory split between user and kernel spaces), affecting the external location of important kernel data structures and symbols. Moreover, different versions of the same OS may have subtle variations for the same kernel-level data structure (e.g., Linux 2.4.20 and 2.6.15-1 have different definitions for *task_struct*). Configuration variation over the same OS (e.g., PAE or swap partition support in modern OSes such as Windows and Linux) adds additional complexity to VM semantic view reconstruction. All such variations should be reflected in the VMwatcher implementation. However, the guest view casting methodology remains effective despite these variations, as confirmed by our evaluation results in Section 4.

3.2.3 *Guest System Call Reconstruction.* With the low-level system call context information (Section 3.1), guest system call reconstruction will derive the detailed semantic information about the system call. The reconstruction leverages the guest view casting technique but using system call semantics as the casting template. In the following description, we assume Linux as the guest OS.

To identify the process that makes the system call, we follow the ESP register value, captured by VMwatcher upon the execution of the system call instruction, to the location of OS data structure *task_struct*. More specifically, we follow the same procedure extensively used in the Linux kernel to identify the current process (i.e., *current = ESP&(8192-1)*). The Linux kernel consolidates two kernel data structures – the kernel stack and the process control block *task_struct* – for the current process in two consecutive pages (8192 bytes). After locating the content of *task_struct*, the information about the current process can be derived by guest view casting using the definition of *task_struct*.

To reconstruct the parameters of the system call, we use the system call convention to interpret the low-level system call context information obtained by VMwatcher: The system call number can be obtained directly from the value of register EAX; while the first six system call arguments are carried by registers EBX, ECX, EDX, ESI, EDI, and EBP. However, the register values alone may not reveal the true semantics of the corresponding arguments – the registers may contain memory addresses pointing to the “true” semantic information (e.g., a file name, an array of command line arguments, or an array of environment settings). To derive the semantic information, we need to follow the register values and locate the corresponding memory contents. The need for such register-guided memory state procurement also arises in another situation: If the system call involves more than six arguments, they will all be pushed to the stack and register EBX will point to the starting address in the stack. In this case, we need to follow EBX to obtain the arguments in the stack.

We note that the above register-guided memory state procurement involves accessing the *virtual* address space of the process making the system call and hence is more complicated than accessing the OS kernel data structures. Fortunately, the memory state reconstruction technique in Section 3.2.2 can be readily applied to perform this task: First, the page table of the current process can be located by following the value of register CR3. For each register that carries a pointer (virtual address) argument of the system call, page table lookup will be performed to map the virtual address to the physical address. Finally, the corresponding memory content can be accessed by the physical address. The astute reader may notice that these steps are performed exactly by VMwatcher routine *vmwatcher_vir_mem_read32* shown in Section 3.2.2.

To reconstruct the return value of the system call, VMwatcher keeps track of the system call and captures its completion, upon which the return value will be obtained by a procedure similar to the one described above for system call argument reconstruction. More specifically, the completion of the system call is indicated by the execution of the instruction right after the system call instruction in the same process. As such, VMwatcher maintains at least the following book-keeping information about each system call: (1) semantic information that has been reconstructed upon the start of the system call (i.e. id of the calling process as well as system call number and parameters), (2) the base address of the calling process' page table, and (3) the (virtual) address of the instruction right after the system call instruction. This way, the completion of a system call can be captured and unambiguously matched with the start of the same system call – despite the possibility of context switches during system call handling.

In the current VMwatcher prototype, guest system call reconstruction has been implemented in both VMware and QEMU VMM platforms, indicating the generality of the guest view casting technique. Our prototype supports real-time capture and reconstruction of 260 Linux system calls.

4. EVALUATION

We evaluate the VMwatcher prototype to demonstrate the three new malware detection and monitoring capabilities (Section 2.3) enabled by VMwatcher. In particular, we show: (1) how VM view comparison effectively detects one of the most stealthy types of malware – rootkits (Section 4.1); (2) how VMwatcher enables "out of the box" deployment of legacy anti-malware systems (Section 4.2); and (3) how VMwatcher supports non-intrusive system call monitoring for malware/attack behavior observation (Section 4.3). Finally, we present performance measurement results (Section 4.4).

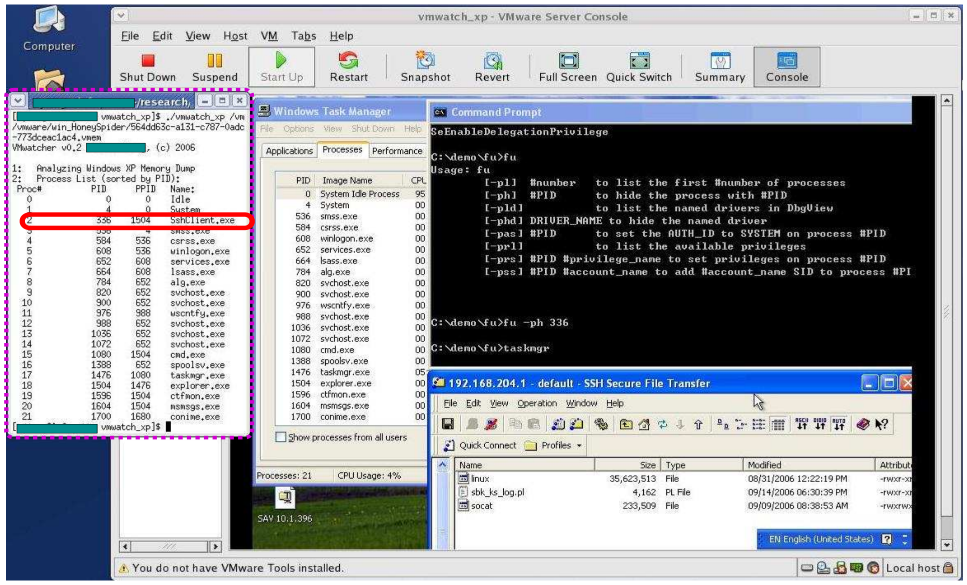
4.1 View Comparison-based Malware Detection

View comparison-based malware detection exploits the very nature of rootkits – hiding attack processes and related files. We have experimented with more than 10 Windows rootkits as well as a dozen Linux rootkits and the view comparison capability is able to detect *all* the rootkits tested and pinpoint the hidden processes and/or files. Due to space constraint, we only present three representative experiments in detail.

Experiment I – view comparison on volatile VM states The first experiment involves a Windows kernel-level FU rootkit [FU 2005]. Figure 3 shows the screenshot of an infected system where the FU rootkit runs and hides a process with PID 336. The system is based on VMware while the host OS is Scientific Linux 4.4 and the guest OS is Windows XP with SP2. In the figure, the background GUI screen with the Windows command shell window shows the *inside* of the VM while the foreground screen (encapsulated with a dashed box) on the left shows the VMwatcher-generated *external* view of the running processes in the same VM.

From the figure, we can observe that a Window command shell (PID: 1080) is created and it is used to invoke the FU rootkit to hide process 336. This hidden process is a running SSH client program – SSH Secure File Transfer (version 3.2.9). This screenshot also shows a short help message on how to invoke the FU rootkit as well as current Windows Task Manager output. The Windows Task Manager does not list the SSH client process, indicating that this (running) process has been successfully hidden.

In comparison, the hidden process is exposed by the external view generated by VMwatcher: The small box with solid lines inside the foreground screen highlights the *SshClient.exe*


vmwatch_xp - VMware Server Console
File Edit View Host VM Tabs Help
Shut Down Suspend Start Up Restart Snapshot Revert Full Screen Quick Switch Summary Console
Computer
[redacted]/research,
You do not have VMware Tools installed.
Local host

```
[[redacted] vmwatch_xp]$ ./vmwatch_xp /vm
/vmware/win_HoneySpider/564dd63c-a131-c787-0adc
-773dceac1ac4.vmem
VMwatcher v0.2 [redacted], (c) 2006

1:   Analyzing Windows XP Memory Dump
2:   Process List (sorted by PID):
 Proc#          PID      PPID    Name:
   0              0         0    Idle
   1              4         0    System
   2            336      1504    SshClient.exe
   3            536         4    smss.exe
   4            584       536    csrss.exe
   5            608       536    winlogon.exe
   6            652       608    services.exe
   7            664       608    lsass.exe
   8            784       652    alg.exe
   9            820       652    svchost.exe
  10            900       652    svchost.exe
  11            976       988    wscntfy.exe
  12            988       652    svchost.exe
  13           1036       652    svchost.exe
  14           1072       652    svchost.exe
  15           1080      1504    cmd.exe
  16           1388       652    spoolsv.exe
  17           1476      1080    taskmgr.exe
  18           1504      1476    explorer.exe
  19           1596      1504    ctfmon.exe
  20           1604      1504    msmsgs.exe
  21           1700      1680    conime.exe
[[redacted] vmwatch_xp]$
```

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance

PID	Image Name	CPL
0	System Idle Process	95
4	System	00
536	smss.exe	00
584	csrss.exe	00
608	winlogon.exe	00
652	services.exe	00
664	lsass.exe	00
784	alg.exe	00
820	svchost.exe	00
900	svchost.exe	00
976	wscntfy.exe	00
988	svchost.exe	00
1036	svchost.exe	00
1072	svchost.exe	00
1080	cmd.exe	00
1388	spoolsv.exe	00
1476	taskmgr.exe	05
1504	explorer.exe	00
1596	ctfmon.exe	00
1604	msmsgs.exe	00
1700	conime.exe	00

☐ Show processes from all users

Processes: 21 CPU Usage: 4%

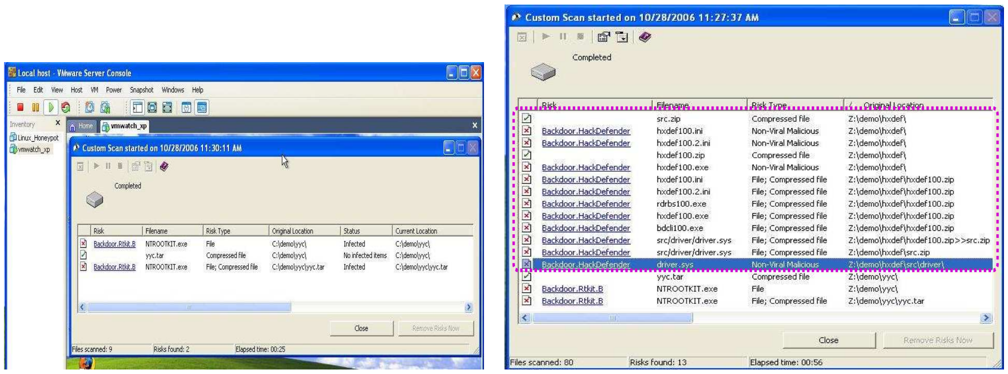
```
SeEnableDelegationPrivilege

C:\demo\fu>fu
Usage: fu
        [-pl]  #number   to list the first #number of processes
        [-ph]  #PID      to hide the process with #PID
        [-pld]           to list the named drivers in DbgView
        [-phd] DRIVER_NAME to hide the named driver
        [-pas] #PID      to set the AUTH_ID to SYSTEM on process #PID
        [-prl]           to list the available privileges
        [-prs] #PID #privilege_name to set privileges on process #PID
        [-pss] #PID #account_name to add #account_name SID to process #PI


C:\demo\fu>fu -ph 336

C:\demo\fu>taskmgr
```

192.168.204.1 - default - SSH Secure File Transfer
File Edit View Operation Window Help
Quick Connect Profiles
EN English (United States)

Name	Size	Type	Modified	Attribut
linux	35,623,513	File	08/31/2006 12:22:19 PM	-rwxr-x
sbk_ks_log.pl	4,162	PL File	09/14/2006 06:30:39 PM	-rwxr-x
socat	233,509	File	09/09/2006 08:38:53 AM	-rwxrwx

Fig. 3. A VMware-based Windows XP VM infected by the FU rootkit

process, which is not shown by the (internal) output of Windows Task Manager. Although we manually conduct this rootkit attack, VMwatcher can be readily adopted by real-world honeypots to detect in-the-wild rootkit attacks. In fact, recent incidents [Rbot 2006] show that the same FU rootkit has been actively used to hide the presence of advanced stealthy bots.


Local host - VMware Server Console
File Edit View Host VM Power Snapshot Windows Help
Inventory
Linux_Honeypot
vmwatch_xp
Home
Custom Scan started on 10/28/2006 11:30:11 AM
Completed
Close
Remove Risks Now

	Risk	Filename	Risk Type	Original Location	Status	Current Location
☒	Backdoor.Rtkit.B	NTROOTKIT.exe	File	C:\demo\yyc\	Infected	C:\demo\yyc\
✓		yyc.tar	Compressed file	C:\demo\yyc\	No infected items	C:\demo\yyc\
☒	Backdoor.Rtkit.B	NTROOTKIT.exe	File; Compressed file	C:\demo\yyc\yyc.tar	Infected	C:\demo\yyc\yyc.tar

Files scanned: 9 Risks found: 2 Elapsed time: 00:25

Custom Scan started on 10/28/2006 11:27:37 AM
Completed
Close
Remove Risks Now

	Risk	Filename	Risk Type	Original Location
✓		src.zip	Compressed file	Z:\demo\hxdef\
☒	Backdoor.HackDefender	hxdef100.ini	Non-Viral Malicious	Z:\demo\hxdef\
☒	Backdoor.HackDefender	hxdef100.2.ini	Non-Viral Malicious	Z:\demo\hxdef\
✓		hxdef100.zip	Compressed file	Z:\demo\hxdef\
☒	Backdoor.HackDefender	hxdef100.exe	Non-Viral Malicious	Z:\demo\hxdef\
☒	Backdoor.HackDefender	hxdef100.ini	File; Compressed file	Z:\demo\hxdef\hxdef100.zip
☒	Backdoor.HackDefender	hxdef100.2.ini	File; Compressed file	Z:\demo\hxdef\hxdef100.zip
☒	Backdoor.HackDefender	rdrbs100.exe	File; Compressed file	Z:\demo\hxdef\hxdef100.zip
☒	Backdoor.HackDefender	hxdef100.exe	File; Compressed file	Z:\demo\hxdef\hxdef100.zip
☒	Backdoor.HackDefender	bdcli100.exe	File; Compressed file	Z:\demo\hxdef\hxdef100.zip
☒	Backdoor.HackDefender	src/driver/driver.sys	File; Compressed file	Z:\demo\hxdef\hxdef100.zip>>src.zip
☒	Backdoor.HackDefender	src/driver/driver.sys	File; Compressed file	Z:\demo\hxdef\src.zip
☒	Backdoor.HackDefender	driver.sys	Non-Viral Malicious	Z:\demo\hxdef\src\driver\
✓		yyc.tar	Compressed file	Z:\demo\yyc\
☒	Backdoor.Rtkit.B	NTROOTKIT.exe	File	Z:\demo\yyc\
☒	Backdoor.Rtkit.B	NTROOTKIT.exe	File; Compressed file	Z:\demo\yyc\yyc.tar

Files scanned: 80 Risks found: 13 Elapsed time: 00:56

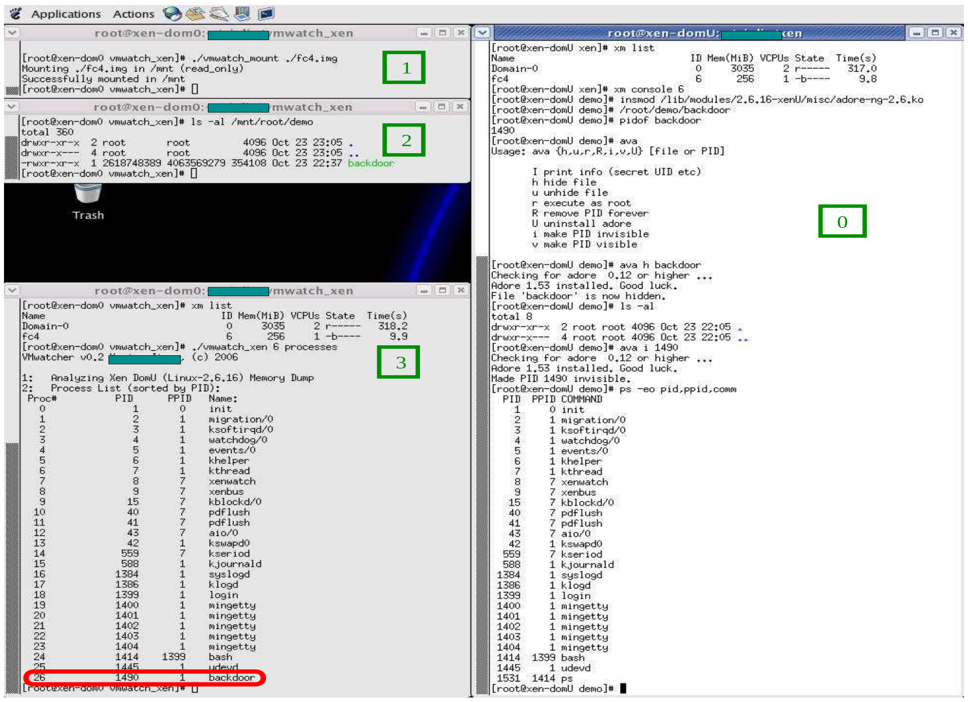
(a) Result of running Symantec AntiVirus inside the VM

(b) Result of running Symantec AntiVirus outside the VM

Fig. 4. Comparison of internal and external views of a hxdef-infected VM

Experiment II – view comparison on persistent VM states In this experiment, we prepare a VMware-based Windows XP VM that contains the files of two rootkits, Hacker Defender (or hxdef) [HackDefender 2004] and NTRootkit [NTRootkit 2005], in the *c:\demo* directory. Both rootkits, when running, are able to hide selected attack files and processes. For evaluation purpose, we only run the hxdef rootkit in the VM. After activating hxdef, we run the Symantec AntiVirus software *inside* the VM and the scanning result is shown in Figure 4(a). The result indicates that the internal view successfully identifies NTRootkit but it *misses* hxdef, because the latter has hidden any file, directory, or process with the string *"hxdef"* in its name.

Meanwhile, we run the same version of Symantec AntiVirus outside of the VM in the host OS and the scanning is based on the VM's semantic view reconstructed by VMwatcher. The result is shown in Figure 4(b). Different from the internal result, the external result catches both NTRootkit and hxdef. The difference is highlighted by the dashed box in Figure 4(b). More importantly, by comparing the two views, we can infer that hxdef, not NTRootkit, is the one that is currently *running*.


Applications
Actions
root@xen-dom0:[redacted]mwatch_xen
1
root@xen-dom0:[redacted]mwatch_xen
2
Trash
root@xen-dom0:[redacted]mwatch_xen
3
root@xen-domU:[redacted]xen
0

```
[root@xen-dom0 vmwatch_xen]# ./vmwatch_mount ./fc4.img
Mounting ./fc4.img in /mnt (read_only)
Successfully mounted in /mnt
[root@xen-dom0 vmwatch_xen]# []
```

```
[root@xen-dom0 vmwatch_xen]# ls -al /mnt/root/demo
total 360
drwxr-xr-x  2 root       root          4096 Oct 23 23:05 .
drwxr-x---  4 root       root          4096 Oct 23 23:05 ..
-rwxr-xr-x  1 2618748389 4063569279 354108 Oct 23 22:37 backdoor
[root@xen-dom0 vmwatch_xen]# []
```

```
[root@xen-dom0 vmwatch_xen]# xm list
Name                                      ID Mem(MiB) VCPUs State   Time(s)
Domain-0                                   0     3035     2 r-----    318.2
fc4                                        6      256     1 -b----      9.9
[root@xen-dom0 vmwatch_xen]# ./vmwatch_xen 6 processes
VMwatcher v0.2 [redacted], (c) 2006

1:   Analyzing Xen DomU (Linux-2.6.16) Memory Dump
2:   Process List (sorted by PID):
 Proc#          PID       PPID   Name:
   0              1          0   init
   1              2          1   migration/0
   2              3          1   ksoftirqd/0
   3              4          1   watchdog/0
   4              5          1   events/0
   5              6          1   khelper
   6              7          1   kthread
   7              8          7   xenwatch
   8              9          7   xenbus
   9             15          7   kblockd/0
  10             40          7   pdflush
  11             41          7   pdflush
  12             43          7   aio/0
  13             42          1   kswapd0
  14            559          7   kseriod
  15            588          1   kjournald
  16           1384          1   syslogd
  17           1386          1   klogd
  18           1399          1   login
  19           1400          1   mingetty
  20           1401          1   mingetty
  21           1402          1   mingetty
  22           1403          1   mingetty
  23           1404          1   mingetty
  24           1414       1399   bash
  25           1445          1   udevd
  26           1490          1   backdoor
[root@xen-dom0 vmwatch_xen]# []
```

```
[root@xen-domU xen]# xm list
Name                                      ID Mem(MiB) VCPUs State   Time(s)
Domain-0                                   0     3035     2 r-----    317.0
fc4                                        6      256     1 -b----      9.8
[root@xen-domU xen]# xm console 6
[root@xen-domU demo]# insmod /lib/modules/2.6.16-xenU/misc/adore-ng-2.6.ko
[root@xen-domU demo]# /root/demo/backdoor
[root@xen-domU demo]# pidof backdoor
1490
[root@xen-domU demo]# ava
Usage: ava {h,u,r,R,i,v,U} [file or PID]

       I print info (secret UID etc)
       h hide file
       u unhide file
       r execute as root
       R remove PID forever
       U uninstall adore
       i make PID invisible
       v make PID visible

[root@xen-domU demo]# ava h backdoor
Checking for adore  0.12 or higher ...
Adore 1.53 installed. Good luck.
File 'backdoor' is now hidden.
[root@xen-domU demo]# ls -al
total 8
drwxr-xr-x  2 root root 4096 Oct 23 22:05 .
drwxr-x---  4 root root 4096 Oct 23 22:05 ..
[root@xen-domU demo]# ava i 1490
Checking for adore  0.12 or higher ...
Adore 1.53 installed. Good luck.
Made PID 1490 invisible.
[root@xen-domU demo]# ps -eo pid,ppid,comm
  PID  PPID COMMAND
    1     0 init
    2     1 migration/0
    3     1 ksoftirqd/0
    4     1 watchdog/0
    5     1 events/0
    6     1 khelper
    7     1 kthread
    8     7 xenwatch
    9     7 xenbus
   15     7 kblockd/0
   40     7 pdflush
   41     7 pdflush
   43     7 aio/0
   42     1 kswapd0
  559     7 kseriod
  588     1 kjournald
 1384     1 syslogd
 1386     1 klogd
 1399     1 login
 1400     1 mingetty
 1401     1 mingetty
 1402     1 mingetty
 1403     1 mingetty
 1404     1 mingetty
 1414  1399 bash
 1445     1 udevd
 1531  1414 ps
[root@xen-domU demo]# █
```

Fig. 5. A Xen-based Fedora Core 4 VM infected by the adore-ng rootkit

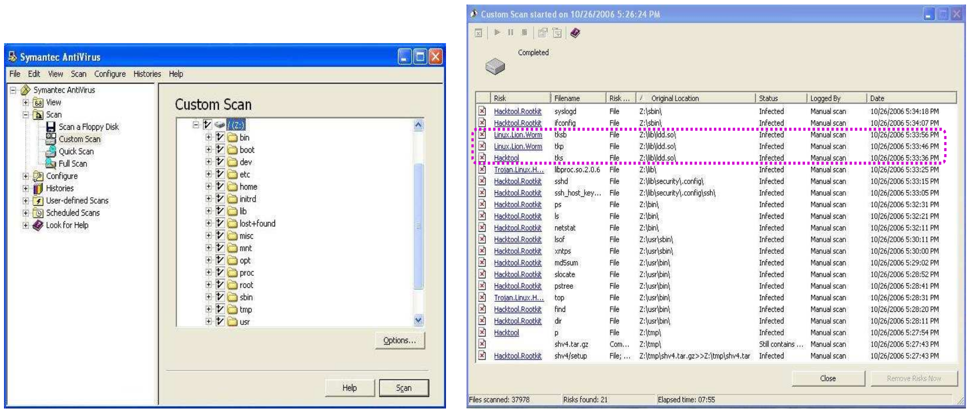
Experiment III – view comparison on both volatile and persistent VM states We describe our experiment with the *adore-ng* [Adoreng 2004] rootkit – an advanced Linux kernel rootkit that will directly replace certain kernel-level function pointers to hide files and processes.

Figure 5 is a screenshot showing an *adore-ng* rootkit that infects a Xen-based Fedora Core 4 VM. There are four numbered *xterm* windows in the figure. The *xterm* window with the number 0 on the right shows the inside of the VM, where the *adore-ng* kernel-level module (LKM) is first loaded (*insmod /lib/modules/2.6.16-xenU/misc/adore-ng-2.6.ko*). A user-level program called *ava* is used to control the LKM’s functionality. Then, a *backdoor* daemon is executed (*/root/demo/backdoor*). After that, *adore-ng* is instructed to conceal the existences of any local files named “backdoor” (*ava h backdoor*) and the backdoor daemon with PID 1490 (*ava i 1490*). As revealed in the same xterm window, the outputs of commands *ls* and *ps* are already manipulated to conceal the existence of any file with the name “backdoor” and any process with PID 1490.

The external view of the VM is shown on the left side of Figure 5. In particular, *xterm* window 2 lists the files under the directory */root/demo/* in the VM; while *xterm* window 3 enumerates current running processes inside the VM. From *xterm* window 2, the internally-concealed backdoor file is *visible* to VMwatcher. Similarly, *xterm* window 3 highlights the internally-hidden “backdoor” process with PID 1490. This experiment further demonstrates that the semantic view reconstructed by VMwatcher cannot be manipulated by the rootkit running inside the VM. As such, view comparison effectively exposes the rootkit’s existence (even if the exposed file and process have unsuspected names).

Software	VMM	Guest OS	Host OS
Symantec AntiVirus 10.1.0396	VMware Server 1.0.1 build-29996	Windows XP (SP2)	Windows XP (SP2)
		Red Hat 7.2, 8.0, 9.0 FC1, 2, 3, 4, RHEL4	
Windows Defender (1.1.1592.0)	VMware Server 1.0.1 build-29996	Windows XP (SP2)	Windows XP (SP2)
Malicious Software Removal Tool 1.2		Red Hat 7.2, 8.0, 9.0 FC1, 2, 3, 4, RHEL4	
Trend Micro ServerProtect for Linux 2.5	Xen 3.0.2-2	Red Hat FC4	Scientific Linux 4.4
	VMware Server 1.0.1 build-29996	Windows XP (SP2)	
		Red Hat 7.2, 8.0, 9.0 FC1, 2, 3, 4, RHEL4	
Kaspersky Anti-Virus 5.5 (trial version)	Xen 3.0.2-2	Red Hat FC4	Scientific Linux 4.4
	VMware Server 1.0.1 build-29996	Windows XP (SP2)	
		Red Hat 7.2, 8.0, 9.0 FC1, 2, 3, 4, RHEL4	
F-Secure Anti-Virus 5.20 Build 5050	Xen 3.0.2-2	Red Hat FC4	Scientific Linux 4.4
	VMware Server 1.0.1 build-29996	Windows XP (SP2)	
		Red Hat 7.2, 8.0, 9.0 FC1, 2, 3, 4, RHEL4	
Frisk F-PROT AntiVirus For Linux 4.6.6	Xen 3.0.2-2	Debian 3.1	Scientific Linux 4.4
	QEMU 0.8.2	Red Hat 7.2, 8.0, 9.0	
McAfee VirusScan 4.24.0	UML 2.4.24	Red Hat 7.2, 8.0, 9.0	Red Hat (RHEL4)
Sophos Anti-Virus 4.05.0	QEMU 0.8.2	Red Hat 7.2, 8.0, 9.0	Red Hat (RHEL4)
Tripwire 4.05.0 (Open Source)	UML 2.4.24	Red Hat 7.2, 8.0, 9.0	Red Hat (RHEL4)
ClamAV 0.88.5 (Open Source)	UML 2.4.24	Red Hat 7.2, 8.0, 9.0	Red Hat (RHEL4)

Table II. A list of off-the-shelf anti-virus software systems deployed on VMwatcher.



Risk	Filename	Risk ...	Original Location	Status	Logged By	Date
Hacktool.Rootkit	syslogd	File	Z:\sbin\	Infected	Manual scan	10/26/2006 5:34:18 PM
Hacktool.Rootkit	ifconfig	File	Z:\sbin\	Infected	Manual scan	10/26/2006 5:34:07 PM
Linux.Lion.Worm	tksb	File	Z:\lib\ldd.so\	Infected	Manual scan	10/26/2006 5:33:56 PM
Linux.Lion.Worm	tkp	File	Z:\lib\ldd.so\	Infected	Manual scan	10/26/2006 5:33:46 PM
Hacktool	tks	File	Z:\lib\ldd.so\	Infected	Manual scan	10/26/2006 5:33:36 PM
Trojan.Linux.H...	libproc.so.2.0.6	File	Z:\lib\	Infected	Manual scan	10/26/2006 5:33:25 PM
Hacktool.Rootkit	sshd	File	Z:\lib\security\.config\	Infected	Manual scan	10/26/2006 5:33:15 PM
Hacktool.Rootkit	ssh_host_key...	File	Z:\lib\security\.config\ssh\	Infected	Manual scan	10/26/2006 5:33:05 PM
Hacktool.Rootkit	ps	File	Z:\bin\	Infected	Manual scan	10/26/2006 5:32:31 PM
Hacktool.Rootkit	ls	File	Z:\bin\	Infected	Manual scan	10/26/2006 5:32:21 PM
Hacktool.Rootkit	netstat	File	Z:\bin\	Infected	Manual scan	10/26/2006 5:32:11 PM
Hacktool.Rootkit	lsof	File	Z:\usr\sbin\	Infected	Manual scan	10/26/2006 5:30:11 PM
Hacktool.Rootkit	xntps	File	Z:\usr\sbin\	Infected	Manual scan	10/26/2006 5:30:00 PM
Hacktool.Rootkit	md5sum	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:29:02 PM
Hacktool.Rootkit	slocate	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:28:52 PM
Hacktool.Rootkit	pstree	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:28:41 PM
Trojan.Linux.H...	top	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:28:31 PM
Hacktool.Rootkit	find	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:28:20 PM
Hacktool.Rootkit	dir	File	Z:\usr\bin\	Infected	Manual scan	10/26/2006 5:28:11 PM
Hacktool	p	File	Z:\tmp\	Infected	Manual scan	10/26/2006 5:27:54 PM
	shv4.tar.gz	Com...	Z:\tmp\	Still contains ...	Manual scan	10/26/2006 5:27:43 PM
Hacktool.Rootkit	shv4/setup	File; ...	Z:\tmp\shv4.tar.gz>>Z:\tmp\shv4.tar	Infected	Manual scan	10/26/2006 5:27:43 PM

Files scanned: 37978   Risks found: 21   Elapsed time: 07:55

(a) A screenshot of Symantec AntiVirus before its scanning

(b) A screenshot of Symantec AntiVirus after scanning

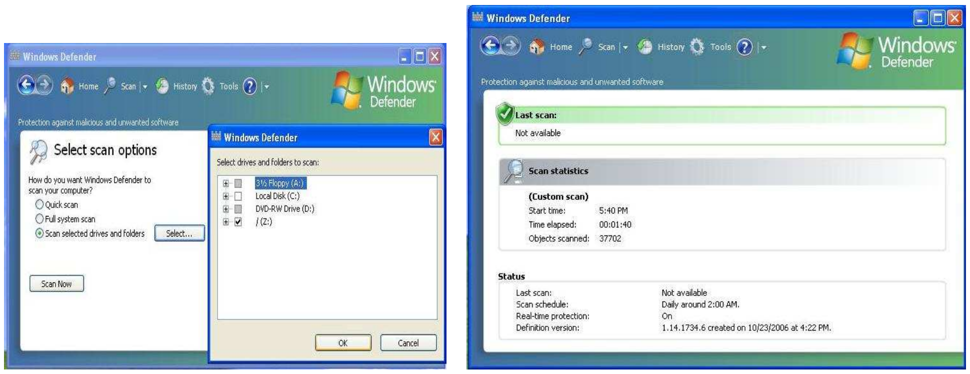
Fig. 6. External inspection of a honeypot using Symantec AntiVirus (version 10.1.0.396)

4.2 “Out of the Box” Deployment of Legacy Anti-Malware Software

VMwatcher also supports “out of the box” deployment and execution of a number of off-the-shelf anti-malware systems and naturally enables the new capability of cross-platform malware detection. We have successfully experimented with 11 real-world anti-virus software systems shown in Table II. For each anti-virus system, Table II also summarizes the corresponding evaluation environment, i.e., the VMM, the host OS, and the guest OS.

In the following, we describe an experiment that deploys the Symantec AntiVirus software (Windows version) “out of the box” to detect malware instances inside a Linux honeypot VM.

Experiment IV – cross-platform malware detection The Linux honeypot is a VMware-based Red Hat 7.2 VM that contains a number of remotely exploitable vulnerabilities. We run Symantec AntiVirus (version 10.1.0.396) in the Windows host domain to detect possible infections inside the honeypot. Figure 6 shows two screenshots of the same Symantec AntiVirus software (version 10.1.0.396): one *before* its scanning and one *after* the scanning. More specifically, Figure 6(a) shows that the corresponding virtual disk of the honeypot VM is externally interpreted and transparently represented as a local “Z:” drive; while Figure 6(b) reports 21 infected files in the VM. Among the infected files, there is a rootkit named SHv4 [Miller 2003], which replaces a number of system-wide utility commands (e.g., ps, ls, ifconfig, netstat, and syslogd) with its own. We also notice that there is a Lion worm [Lion 2001] infection in the report (highlighted in the dashed box of Figure 6(b)), which we believe is misclassified. The two Lion-infected files identified are *tksb* and *tkp* under directory */lib/ldd.so*. It turns out that *tksb* is a shell script that functions as a log cleaner, while *tkp* is a Perl script essentially looking for user names and passwords in collected network traffic. Later forensic analysis reveals that an attacker first exploited the Apache web server vulnerability [Apache 2003] to gain system access. After that, he exploited the local ptrace kernel vulnerability [Secunia 2003] to escalate his privilege to



Select scan options

How do you want Windows Defender to scan your computer?

( ) Quick scan
( ) Full system scan
[✓ Scan selected drives and folders] Select...

Scan Now

Windows Defender

Select drives and folders to scan:

- ☐ 3½ Floppy (A:)
- ☐ Local Disk (C:)
- ☐ DVD-RW Drive (D:)
- ☑ / (Z:)

OK Cancel

Last scan:
Not available

Scan statistics

(Custom scan)

Start time:	5:40 PM
Time elapsed:	00:01:40
Objects scanned:	37702

Status

Last scan:	Not available
Scan schedule:	Daily around 2:00 AM.
Real-time protection:	On
Definition version:	1.14.1734.6 created on 10/23/2006 at 4:22 PM.

(a) A screenshot of Windows Defender before its scanning

(b) A screenshot of Windows Defender after scanning

Fig. 7. External inspection of a honeypot using Microsoft Windows Defender (version 1.1.1592.0)

system root before installing the SHv4 rootkit.

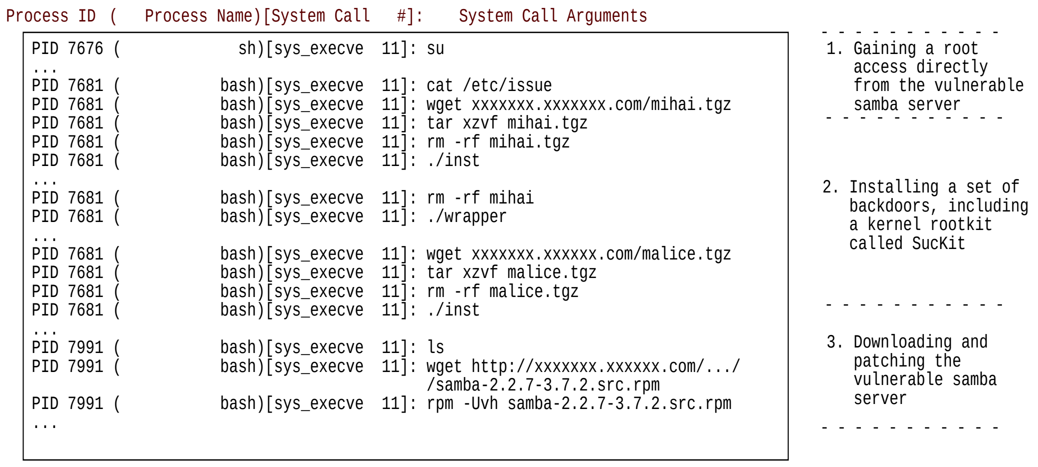
For comparison, we also run Microsoft Windows Defender (version 1.1.1592.0) in the host domain of the same compromised VM and the result is shown in Figure 7. Interestingly, the scanning result shows no malware infection. It seems that this specific version of anti-virus software targets malware on Windows platforms thus missing the Linux malware.

4.3 Non-intrusive Malware/Attack Monitoring

To demonstrate the non-intrusive system call monitoring capability of VMwatcher, we deploy VMwatcher in an integrated, VM-based malware experimentation platform we developed earlier. The platform consists of a front-end and a back-end: The front-end is a honeyfarm [Jiang and Xu 2004] that captures real-world malware and intrusion incidents from production networks, while the back-end is a confined and high-fidelity "playground" [Jiang et al. 2005] where real-world malware can be unleashed and closely observed. Previously, we installed Sebek [Sebek 2006] – a widely used honeypot monitoring and logging tool – inside the front-end and back-end VMs. Unfortunately, Sebek can be detected, disabled, or bypassed by counter-monitoring techniques such as unsebek [Corey 2004] and NoSEBrEaK [Dornseif et al. 2004], which can be easily adopted by stealthy malware in the wild. In comparison, VMwatcher is less susceptible to counter-monitoring as it runs outside without any "in the box" entity and thus is entirely passive. With VMwatcher, we can now remove Sebek from the VMs.

In the following, we describe two experiments to demonstrate the effectiveness of VMwatcher in monitoring and logging malware and attack behavior in a non-intrusive, tamper-resistant way.

Experiment V - non-intrusive monitoring of attack behavior This attack incident, captured by the VMwatcher-enabled front-end honeyfarm, is against the vulnerable Samba



```
Process ID (   Process Name)[System Call   #]:    System Call Arguments

PID 7676 (            sh)[sys_execve  11]: su
...
PID 7681 (          bash)[sys_execve  11]: cat /etc/issue
PID 7681 (          bash)[sys_execve  11]: wget xxxxxxx.xxxxxxx.com/mihai.tgz
PID 7681 (          bash)[sys_execve  11]: tar xzvf mihai.tgz
PID 7681 (          bash)[sys_execve  11]: rm -rf mihai.tgz
PID 7681 (          bash)[sys_execve  11]: ./inst
...
PID 7681 (          bash)[sys_execve  11]: rm -rf mihai
PID 7681 (          bash)[sys_execve  11]: ./wrapper
...
PID 7681 (          bash)[sys_execve  11]: wget xxxxxxx.xxxxxx.com/malice.tgz
PID 7681 (          bash)[sys_execve  11]: tar xzvf malice.tgz
PID 7681 (          bash)[sys_execve  11]: rm -rf malice.tgz
PID 7681 (          bash)[sys_execve  11]: ./inst
...
PID 7991 (          bash)[sys_execve  11]: ls
PID 7991 (          bash)[sys_execve  11]: wget http://xxxxxxx.xxxxxx.com/.../
                                           /samba-2.2.7-3.7.2.src.rpm
PID 7991 (          bash)[sys_execve  11]: rpm -Uvh samba-2.2.7-3.7.2.src.rpm
...
```

```
- - - - - - - - - - -
1. Gaining a root
   access directly
   from the vulnerable
   samba server
- - - - - - - - - - -

2. Installing a set of
   backdoors, including
   a kernel rootkit
   called SucKit

- - - - - - - - - - -

3. Downloading and
   patching the
   vulnerable samba
   server
- - - - - - - - - - -
```

Fig. 8. VMwatcher log excerpt showing attacker behavior after breaking into Samba server

server (version 2.2.1a-4). The honeypot VM was deployed at 21:00PM, October 7th, 2007 and compromised 2 hours later. VMwatcher captured and logged all system call events during and after the attacker's exploitation. The system call log contains sufficient details to understand the attacker's exploitation steps and post-exploitation behavior. Figure 8 shows an excerpt of the log that records the commands executed by the attacher via the shell (bash) spawned from the exploitation.

From Figure 8, we observe that, after the successful exploitation, the attacker gained root privilege directly from the vulnerable Samba server and then downloaded and installed a set of pre-packaged tools (*mihai.tgz* and *malice.tgz*). Later forensic analysis shows that the tools include an infamous rootkit named SucKit [SucKit 2001] and a trojaned sshd daemon. Interestingly, the attacker used the trojaned sshd daemon (which spawned another shell with PID 7991) to upgrade and patch the vulnerable Samba server, so that no other malware or attackers can infiltrate the server through the same vulnerability.

Experiment VI - non-intrusive monitoring of Lion worm behavior This experiment with the Lion worm [Lion 2001] is performed in the back-end malware playground enabled by VMwatcher. Figure 9 shows a log excerpt from a VMwatcher-enabled VM in the playground. The Lion worm exploits a vulnerable version of the DNS server and the VMwatcher log reveals details of the worm's behavior after compromising the DNS server.

More specifically, the Lion worm first redirects standard input, output and error to an open file descriptor 6 – a network socket (by making *sys_dup2* system call) and creates a shell (*sys_execve*). After the redirection, the compromised named process will get the attacker's instructions directly from the established network connection and put all output to the same network connection. The Lion worm then reads local password files */etc/passwd* and */etc/shadow*, removes the system-wide log data (through the three *bin/rm* commands), replicates itself in the compromised VM (*/usr/bin/lynx*), and executes (*./lion*). From the VMwatcher log, we also observe that the Lion worm attempts to replace all *index.html* files in the compromised VM with its own copy, defacing the corresponding web page.

```
Process ID (   Process Name)[System Call   #]:    System Call Arguments
PID 31122(          named)[sys_accept  102]5: socket 23
PID 31122( syscall return)[            102]5: 6
PID 31122(          named)[sys_recvfro 102]12: socket 22
PID 31122(          named)[sys_sendto  102]11: socket 22
..
PID 31122(          named)[sys_dup2     63]: oldfd 6; newfd 2
PID 31122(          named)[sys_dup2     63]: oldfd 6; newfd 1
PID 31122(          named)[sys_dup2     63]: oldfd 6; newfd 0
PID 31122(          named)[sys_execve   11]: /bin/sh
...
PID 31122(             sh)[sys_fork      2]:
PID 31122( syscall return)[              2]: 31168
PID 31168(             sh)[sys_execve   11]: /bin/rm -rf /dev/.lib
PID 31169(             sh)[sys_execve   11]: /bin/mkdir /dev/.lib
PID 31170(             sh)[sys_execve   11]: /usr/bin/killall -HUP inetd
PID 31171(             sh)[sys_execve   11]: /sbin/ifconfig -a
PID 31172(             sh)[sys_execve   11]: /bin/cat /etc/passwd
PID 31173(             sh)[sys_execve   11]: /bin/cat /etc/shadow
PID 31174(             sh)[sys_execve   11]: /bin/rm -fr 1i0n
PID 31175(             sh)[sys_execve   11]: /bin/rm -fr /.bash_history
PID 31176(             sh)[sys_execve   11]: /bin/rm -rf /var/log/maillog
PID 31177(             sh)[sys_execve   11]: /bin/chmod 755 lion
PID 31178(             sh)[sys_execve   11]: /usr/bin/lynx -source http://192.168.2.2:27374
...
PID 31180(             sh)[sys_execve   11]: ./lion
PID 31181(             sh)[sys_execve   11]: /usr/bin/nohup find / -name index.html -exec /bin/cp index.html {} ;
PID 31181(             sh)[sys_execve   11]: /bin/nice -5 -- find / -name index.html -exec /bin/cp index.html {} ;
PID 31181(             sh)[sys_execve   11]: /sbin/find / -name index.html -exec /bin/cp index.html {} ;
PID 31181(             sh)[sys_execve   11]: /usr/sbin/find / -name index.html -exec /bin/cp index.html {} ;
PID 31181(             sh)[sys_execve   11]: /bin/find / -name index.html -exec /bin/cp index.html {} ;
PID 31181(             sh)[sys_execve   11]: /usr/bin/find / -name index.html -exec /bin/cp index.html {} ;
PID 31183(             sh)[sys_execve   11]: /bin/cp index.html /var/www/html/index.html
...
```

Fig. 9. VMwatcher log excerpt showing Lion worm behavior after compromising a vulnerable DNS server

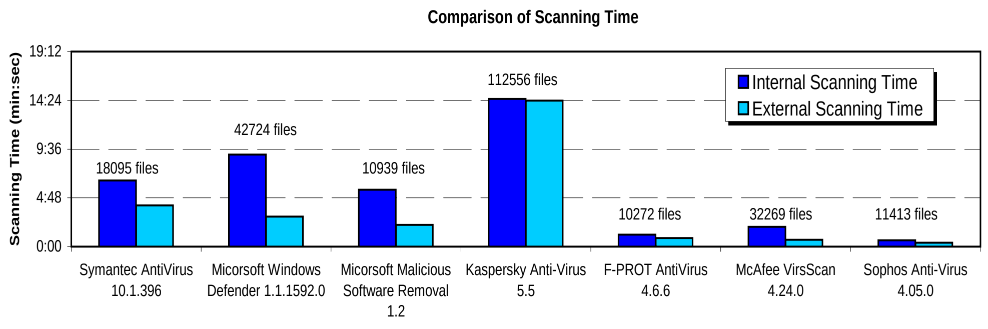


Fig. 10. Comparison of internal scanning and external scanning time (external scanning is enabled by VMwatcher)

4.4 Performance

In this section, we present three sets of performance measurement results.

The first set of experiments is to compare the time to perform internal scanning and external scanning (enabled by VMwatcher) on a VM. In particular, we choose 7 different anti-malware software systems and each system performs an external scan and an internal scan on a particular VM: (1) Symantec AntiVirus, Microsoft Windows Defender, and Microsoft Malicious Software Removal Tool each scan a Windows XP VM (256MB memory and 6GB disk) with the host OS being Windows XP Professional (2GB memory and 120GB disk); (2) Kaspersky Anti-Virus inspects a Red Hat 8.0 VM (1GB memory and 4GB disk) with Scientific Linux 4.4 as the host OS (2GB memory and 180GB disk); (3) F-PROT AntiVirus examines a Debian 3.1 Linux VM based on the Xen VMM while domain 0 is running Scientific Linux 4.4 (4GB memory and 330GB disk); (4) McAfee VirusScan and Sophos Anti-Virus each look into a Red Hat 7.0 VM (128MB memory and 512MB disk) that is running inside a UML VMM. The host OS is RedHat Enterprise Linux 4 with 2GB memory and 135GB disk. The results and the total number of scanned files are shown in Figure 10. It is interesting to notice that the internal scanning session always takes longer time than its external counterpart, a result that sounds counter-intuitive. However, considering the potential disk I/O slowdown introduced by virtualization as well as the availability of larger memory space in the host domain, the shorter external scanning time is actually reasonable. Another reason is that any software-based VM implementation (on the x86 platform) needs to faithfully emulate "privileged" instructions (e.g., STI/CLI) and "sensitive" instructions (e.g., PUSHF/POPF). Such emulation may also have slowed down the internal scanning.

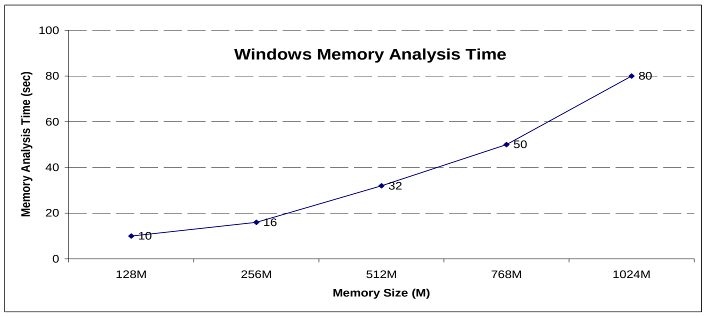


Fig. 11. Time to analyze live raw VM Memory image

The second set of experiments measures the time to analyze a live raw VM memory image by VMwatcher. Note that in the current VMwatcher prototype, we assume that the Windows kernel-level symbols are not available due to its close-source nature while the Linux symbols are available and can be used to speed up the memory semantic view reconstruction. Figure 11 shows the time needed to examine a raw Windows memory image when we vary the memory size from 128MB to 1GB. As expected, the analysis time grows with the size of available memory allocated to the VM. Our results also show that with the availability of Linux symbols, a raw memory analysis session can be finished within 0.5 second, regardless of the size of memory allocated to the VM.

The third set of experiments measures the overhead introduced by real-time interception and reconstruction of guest system call events. Our physical test machine is a Dell PowerEdge 2685 server with a 3.73GHz Intel Xeon processor and 4GB of RAM. The host OS is RedHat Fedora Core 5 with the default 2.6.15-1.2054_FC5 Linux kernel. We evaluate the VMware-based implementation of VMwatcher using two benchmarks: one is the standard UnixBench [UnixBench 2007] micro-benchmark while the another one is an application benchmark using ApacheBench [Apache 2007]. To measure the overhead, we run each benchmark program twice: once with VMwatcher disabled and once with VMwatcher enabled for guest system call reconstruction. Each recorded result is the average of ten runs.

The UnixBench benchmark measures the fine-grained performance impact of VMwatcher and the results are shown in Table III: The worst case overhead is 11.0% while the overheads in most other cases are below 10%. For the Apache experiment, we run the Apache web server in the default Worker MPM mode and run the standard ApacheBench program in another machine connected to the web server via a dedicated switch. The ApacheBench

Benchmark	w/o VMwatcher	w/ VMwatcher	Overhead
Dhrystone	654.0	654.4	0%
Whetstone	311.2	279.9	10.1%
Execl	334.0	297.3	11.0%
File copy 256B	387.8	350.9	9.3%
File copy 1kB	600.9	588.0	2.1%
File copy 4kB	1055.0	1000.14	5.2%
Pipe throughput	269.1	260.8	3.3%
Process creation	245.3	225.2	8.2%
Shell scripts (8)	558.3	537.8	3.7%
System call	198.8	189.5	4.7%

Table III. VMwatcher overhead measurement using UnixBench (for the first two data columns, higher is better)

program stresses the web server with requests for a web page of 32KB and reports the web server throughput. Our experiments show that VMwatcher incurs a 3.8% degradation of web server throughput.

5. DISCUSSION

VMwatcher assumes a trusted VMM layer to isolate untrusted processes inside a monitored VM from affecting VMwatcher. This assumption is needed and there exist parallel efforts in building such trusted VMMs (Section 6). The VMM essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. In the following, we examine possible attacks against VMwatcher.

Guest view subversion Such an attack could be launched from inside the VM by distorting the guest function or state of the VM. The distorted view will then be observed by VMwatcher through virtual machine introspection, resulting in erroneous guest view reconstruction. We further classify the guest view subversion attacks into two types – both will lead to distorted VMM-level observations:

The first type of guest view subversion attacks involve introducing a subverted guest kernel function, which is different from the one used by VMwatcher for semantic view reconstruction. For example, guest system call reconstruction requires the semantics and convention of system calls. Hence it is possible that an attacker *remap* the system calls or system call convention in a non-standard way to mislead or escape VMwatcher. As another example, instead of using the original Linux kernel scheduler with the default all-tasks list to dispatch processes, an advanced malware could implement its own scheduler, which maintains a shadow list of hidden processes without linking them to the all-tasks list. Without knowledge about the subverted scheduler, VMwatcher is not able to accurately identify all running processes. Although it is challenging to understand the details of a subverted guest function, we point out that the subversion behavior itself could be detected. In the previous example, the subversion of the original scheduler code will essentially modify the text segment of the guest kernel and a simple hash calculation (e.g., MD5) will immediately detect that. To counter this type of attacks, VMwatcher can be extended to validate the integrity of guest functions and critical kernel objects (e.g., *sys_call_table* and IDT). Some existing systems such as Livewire [Garfinkel and Rosenblum 2003], Copilot [Petroni et al. 2004], SecVisor [Seshadri et al. 2007] and NICKLE [Riley et al. 2008] can be leveraged for that purpose.

The second type of guest view subversion attacks involve contaminating the guest internal states presented to VMwatcher for semantic view reconstruction. For example, an attacker may choose to manipulate certain guest kernel data structures (e.g., the global process list) to mislead the guest view casting function. The current VMwatcher prototype does require that the guest kernel data structures used by VMwatcher be trusted. Fortunately, there exist solutions that can be leveraged to either ensure the semantic integrity of kernel data structures [Petroni et al. 2006] or prevent the injection of malicious kernel code that manipulates kernel data structures [Riley et al. 2008].

Guest caching exploitation This attack may occur if a modified file is not reflected in time in the disk that is being examined by VMwatcher. One potential result from this attack is that a malware may avoid any file scanning-based detection as it can deliberately hide itself inside the cache without actually committing to the disk. There are two possible counter-measures: one is to make sure that those related guest kernel threads such as *bdflush* and *kupdate* dutifully look for dirty pages and flush them to the disk in time. The second counter-measure is to directly examine the cached contents through VM introspection since the cached contents are still in the volatile memory. However, one challenge here is to seamlessly integrate the memory content with related disk files and present them to the external anti-malware functions.

VM fingerprinting Finally, we note that VM environments can potentially be fingerprinted and detected [Klein 2003; Rutkowska 2004] by attackers. In fact, a number of recent malware instances are able to check whether they are running inside a VM and if so, choose to exhibit different behavior [Agobot 2004]. As a counter-measure, we can improve the fidelity of VM implementation (e.g., as proposed in [Kortchinsky 2004; Liston and Skoudis 2006]) to thwart some of the VM detection schemes. Still, there exist more fundamental attacks (e.g., VM detection based on timing and performance overhead characterization) that are difficult to mitigate. However, from another perspective, as virtualization continues to gain popularity, the concern over VM detection may become less significant as more malware becomes VMM-agnostic with VMs becoming increasingly attractive targets to attackers.

6. RELATED WORK

Enhancing security with virtualization The first area of related work is the use of virtualization technologies to enhance system security. More specifically, leveraging recent advances in virtualization, researchers have adopted VMs to detect intrusions [Garfinkel and Rosenblum 2003; Joshi et al. 2005; Kourai and Chiba 2005], analyze intrusions [Dunlap et al. 2002; Koju et al. 2005], diagnose system problems [King et al. 2005; Whitaker et al. 2004], isolate services [Bryant et al. 2003; Meushaw and Simard 2000], and implement honeypots/honeyfarms [Honeynet 2008; Anagnostakis et al. 2005; Jiang and Xu 2004]. These applications leverage the desirable properties of VMs (e.g., isolation and dynamic configurability) to improve security and accountability of systems without having to trust the guest OS and application programs.

Our work complements or enhances these efforts by elevating the usability of the VM introspection methodology [Garfinkel and Rosenblum 2003], which is pioneered by the Livewire system [Garfinkel and Rosenblum 2003]. VM introspection in Livewire is capable of examining low-level VM states (e.g., disk blocks and memory pages) from outside the VM. However, for the reconstruction of high-level semantic views (e.g., files, processes, and kernel modules), it still needs a new technique, similar to the guest view casting technique in our system, to effectively bridge the semantic gap. While VMwatcher aims at supporting legacy anti-malware software, Livewire mainly supports a specialized IDS built from scratch to detect a more targeted set of intrusions. Furthermore, we propose and demonstrate the opportunity of view comparison for self-hiding malware detection, which is not addressed in [Garfinkel and Rosenblum 2003].

IntroVirt [Joshi et al. 2005] is another closely related work that applies VM introspection to execute vulnerability-specific predicates in a VM for intrusion reproduction. There exist two major differences between IntroVirt and VMwatcher. First, IntroVirt develops a specialized predicate engine that does not aim at accommodating legacy anti-malware systems – a goal achieved by VMwatcher; Second, IntroVirt needs to overwrite a portion of the vulnerable program code with its own predicates or invoke existing code in either guest applications or the guest kernel. Such an approach is considered intrusive and will introduce undesirable perturbation in the VM. Consequently, it needs to resort to taking a checkpoint of the whole VM before making any changes to the VM state and rolling back to the saved checkpoint if perturbance is detected [Joshi et al. 2005]. In contrast, VMwatcher takes a non-intrusive approach and aims at externally reconstructing VM semantic views.

Implementing malware with virtualization Leveraging virtualization technologies, researchers have also demonstrated the potential of implementing virtualization-based malware [King et al. 2006; Rutkowska 2006; Zovi 2006]. King et al. [King et al. 2006] proposes the notion of VM-based rootkit (VMBR) which can be dynamically inserted underneath an existing OS. Rutkowska et al. [Rutkowska 2006] further implements a hardware virtualization-based rootkit prototype called “Blue Pill”, claiming the creation of 100% undetectable malware. Another hardware virtualization-based rootkit – the Vitriol [Zovi 2006] rootkit – independently confirms this significant threat. We point out that these emerging threats can be mitigated or even defeated by recent research efforts in secure booting [Arbaugh et al. 1997] and secure VMMs such as sHype [Sailer et al. 2005] and TRANGO [Trango 2008]. With secure booting, VMMs will maintain the lowest-level access to the system thus preventing them from being subverted. Paralleling these efforts, VMwatcher assumes the non-subvertability of VMMs in anticipation of future deployment of these anti-subversion solutions.

Detecting integrity violations with secure monitors VMwatcher is also related to projects that use secure monitors to detect system integrity violations [Garfinkel et al. 2003; Pennington et al. 2003; Petroni et al. 2006; Petroni et al. 2004]. Copilot [Petroni et al. 2004] detects possible kernel integrity violations by running the monitoring software on a separate PCI card. The monitoring software periodically grabs a copy of the system memory and examines possible integrity violations. A specification-based integrity checker is later proposed [Petroni et al. 2006] to examine the integrity of dynamic kernel data. Note that these two systems only take snapshots of volatile memory states. The storage-based intrusion monitor [Pennington et al. 2003] leverages the isolation provided by a file server (e.g., an NFS server) and independently identifies possible symptoms of malware infections in disk states. Note that it only captures a system’s persistent states. As a result, it is not able to detect elusive malware that may be hiding entirely in the memory (e.g., kernel-level rootkits). In contrast, VMwatcher examines both volatile and persistent states as well as system call events for malware detection and monitoring.

Detecting malware with cross-view comparison The notion of view comparison-based analysis is initially proposed by Wang et al. [Wang et al. 2005] in their Strider GhostBuster system. Their system performs two scans – an internal scan and an external clean scan

– and the two scanning results are then compared for malware detection. However, the external clean scan is done by rebooting the machine being examined with a clean OS (i.e., a WinPE CD). This will, unfortunately, destroy all non-persistent states. On the other hand, VMwatcher performs *live* VM state and system call event procurement and semantic view reconstruction without losing any malware information. A number of recent rootkit detection systems such as RootkitRevealer [RootkitRevealer 2007] and BlackLight [BlackLight 2007] also adopt the same methodology to detect stealthy malware. However, there is a lack of a trustworthy view for comparison as all the views (though from different perspectives) are generated from *inside* the system being monitored.

General intrusion detection techniques Finally, we discuss the general intrusion detection systems (IDS), including the host-based IDS [Kim and Spafford 1994] and the network-based IDS [Snort 2008; Paxson 1999; Weaver et al. 2007]. We note that a network-based IDS is deployed outside of an end-system, achieving high attack resistance at the cost of lower visibility on the internal system states. A traditional host-based IDS runs inside the end-system and is able to directly inspect the states and events of the system, achieving better visibility. However, it sacrifices tamper resistance as it could be compromised during an attack. In contrast, VMwatcher achieves stronger tamper resistance while maintaining high visibility on the system's internal semantic states and system call events.

7. CONCLUSION

We have presented VMwatcher, a VMM-based approach that enables "out of the box" malware detection and monitoring by addressing the semantic gap challenge. More specifically, VMwatcher achieves stronger tamper-resistance by moving anti-malware facilities out of the monitored VM while maintaining the native semantic view of the VM via external semantic view reconstruction. Our evaluation of the VMwatcher prototype demonstrates its practicality and effectiveness. In particular, our experiments with real-world stealthy rootkits and worms further demonstrate the power of the new malware detection and monitoring capabilities enabled by VMwatcher.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous *ACM Transactions on Information and System Security (TISSEC)* reviewers whose comments have greatly helped to improve the presentation of this paper. The anonymous reviewers of a preliminary conference version of this paper [Jiang et al. 2007] are also acknowledged. This work was supported in part by the US National Science Foundation (NSF) under Grants CNS-0716376, CNS-0716444 and CNS-0546173. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Agobot. *http://www.f-secure.com/v-descs/agobot.shtml*.

CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. *http://www.cert.org/advisories/CA-2002-17.html, 2003*.

F-Secure Blacklight. *http://www.f-secure.com/blacklight/*.

FU Rootkit. *http://www.rootkit.com/project.php?id=12*.

hxdef. *http://hxdef.czweb.org*.

NTRootkit. *http://www.megasecurity.org/Tools/Nt_rootkit_all.html*.

Rbot. *http://research.sunbelt-software.com/threatdisplay.aspx?name=Rbot&threatid=14953*.

SANS Institute: Lion worm. *http://www.sans.com/y2k/lion.htm*.

Sebek. *http://www.honeynet.org/tools/sebek/*.

Snort. *http://www.snort.org*.

The adore-ng Rootkit. *http://stealth.openwall.net/rootkits/*.

The Honeynet Project. *http://www.honeynet.org*.

TRANGO, the Real-Time Embedded Hypervisor. *http://www.trango-systems.com/*.

VMware. *http://www.vmware.com/*.

Xen Interface Manual. *http://www.xensource.com/files/xen_interface.pdf, 2004*.

ANAGNOSTAKIS, K. G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., AND KEROMYTIS, A. D. 2005. Detecting Targeted Attacks Using Shadow Honeypots. *Proc. of the 14th USENIX Security Symposium*.

APACHE. 2007. The Apache HTTP Server Project. *http://httpd.apache.org*.

ARBAUGH, W. A., FARBERT, D. J., AND SMITH, J. M. 1997. A Secure and Reliable Bootstrap Architecture. *Proc. of the 1997 IEEE Symposium on Security and Privacy*.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., A. HO, R. N., PRATT, I., AND WARFIELD, A. 2003. Xen and the Art of Virtualization. *Proc. of the 19th ACM Symposium on Operating Systems Principles*.

BELLARD, F. 2005. QEMU, a Fast and Portable Dynamic Translator. *Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track)*.

BELLARD, F. 2006. QEMU Accelerator User Documentation. *http://fabrice.bellard.free.fr/qemu/kqemu-doc.html*.

BRYANT, E., EARLY, J., GOPALAKRISHNA, R., ROTH, G., SPAFFORD, E. H., WATSON, K., WILLIAMS, P., AND YOST, S. 2003. Poly2 Paradigm: A Secure Network Service Architecture. *Proc. of the 19th Annual Computer Security Applications Conference*.

BUGCHECK. 2006. GREPEXEC: Grepping Executive Objects from Pool Memory. *http://www.uninformed.org/?v=4&a=2&t=sumry*.

CHEN, P. M. AND NOBLE, B. D. 2001. When Virtual is Better Than Real. *HotOS VIII, Schoss Elmau, Germany*.

COREY, J. 2004. Local Honeypot Identification. *Phrack 62:article 07 of 15*.

DIKE, J. 2002. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

DORNSEIF, M., HOLZ, T., AND KLEIN, C. 2004. NoSEBrEaK - Attacking Honeynets. *Proc. of the 5th Annual IEEE Information Assurance Workshop, Westpoint*.

DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *5th Symposium on Operating Systems Design and Implementation (OSDI)*.

GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. 2003. Terra: A Virtual Machine-Based Platform for Trusted Computing. *Proc. of the 2003 Symposium on Operating Systems Principles (SOSP)*.

GARFINKEL, T. AND ROSENBLUM, M. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proc. of the 2003 Network and Distributed System Security Symposium*.

JIANG, X., WANG, X., AND XU, D. 2007. Stealthy Malware Detection Through VMM-Based “Out-of-the-Box” Semantic View Reconstruction. *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*.

JIANG, X. AND XU, D. 2004. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proc. of the 13th USENIX Security Symposium*.

JIANG, X., XU, D., WANG, H. J., AND SPAFFORD, E. H. 2005. Virtual Playgrounds for Worm Behavior Investigation. *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*.

JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2005. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. *Proc. of the 2005 Symposium on Operating Systems Principles (SOSP)*.

KIM, G. H. AND SPAFFORD, E. H. 1994. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. *In Systems Administration, Networking and Security Conference III, USENIX*.

KING, S. T. AND CHEN, P. M. 2003. Backtracking Intrusions. *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*.

KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. 2006. SubVirt: Implementing Malware with Virtual Machines. *Proc. of the 2006 IEEE Symposium on Security and Privacy*.

KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2005. Debugging Operating Systems with Time-Traveling Virtual Machines. *Proc. of the 2005 Annual USENIX Technical Conference*.

KLEIN, T. 2003. Scooby Doo - VMware Fingerprint Suite. *http://www.trapkit.de/research/vmm/scoopydoo/index.html*.

KOJU, T., TAKADA, S., AND DOI, N. 2005. An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*.

KORTCHINSKY, K. 2004. Honeypots: Counter measures to VMware fingerprinting. *http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html*.

KOURAI, K. AND CHIBA, S. 2005. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*.

LISTON, T. AND SKOUDIS, E. 2006. On the Cutting Edge: Thwarting Virtual Machine Detection. *http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf*.

MEUSHAW, R. AND SIMARD, D. 2000. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*.

MICROSOFT. 2003. Volume Shadow Copy Service. *http://technet2.microsoft.com/WindowsServer/en/library/2b0d2457-b7d8-42c3-b6c9-59c145b7765f1033.mspx?mfr=true*.

MILLER, J. V. 2003. SHV4 Rootkit Analysis. *https://tms.symantec.com/members/AnalystReports/030929-Analysis-SHV4Rootkit.pdf*.

PAXSON, V. 1999. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks, 31(23-24):2345-2463*.

PENNINGTON, A. G., STRUNK, J. D., GRIFFIN, J. L., SOULES, C. A. N., GOODSON, G. R., AND GANGER., G. R. 2003. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. *Proc. of the 12th USENIX Security Symposium*.

PETRONI, N., FRASER, T., WALTERS, A., AND ARBAUGH, W. 2006. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. *Proc. of the 15th USENIX Security Symposium*.

PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. 2004. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *Proc. of the 13th USENIX Security Symposium*.

PROVOS, N. 2003. Improving Host Security with System Call Policies. *Proc. of the 12th USENIX Security Symposium*.

RILEY, R., JIANG, X., AND XU, D. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*.

ROOTKITREVEALER. 2007. RootkitRevealer. *http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx*.

RUTKOWSKA, J. 2004. Red Pill: Detect VMM using (almost) One CPU Instruction. *http://invisiblethings.org/papers/redpill.html*.

RUTKOWSKA, J. 2006. Subverting Vista Kernel For Fun And Profit. *Blackhat 2006*.

SAILER, R., VALDEZ, E., JAEGER, T., PEREZ, R., VAN DOORN, L., GRIFFIN, J. L., AND BERGER, S. 2005. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. *IBM Research Report RC23511*.

The SucKit Rootkit. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, December 2001.

SECUNIA. 2003. Linux Kernel Ptrace Privilege Escalation Vulnerability. *http://www.secunia.com/advisories/8337/*.

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

UNIXBENCH. 2007. UnixBench. *http://www.tux.org/pub/tux/benchmarks/System/unixbench*.

WANG, Y.-M., BECK, D., VO, B., ROUSSEV, R., AND VERBOWSKI, C. 2005. Detecting Stealth Software with Strider GhostBuster. *Proc. of the 2005 International Conference on Dependable Systems and Networks*.

WEAVER, N., PAXSON, V., AND GONZALEZ, J. 2007. The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. *Proc. of the FPGA 2007*.

WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. 2004. Configuration Debugging as Search: Finding the Needle in the Haystack. *Proc. of USENIX OSDI 2004*.

ZOVI, D. D. 2006. Hardware Virtualization Based Rootkits. *Blackhat 2006*.

Received February 2008; revised June 2008; accepted XXXX