CERIAS Tech Report 2007-80
Stealthy Malware Detection Through VMM-Based
by Xuxian Jiang, Xinyuan Wang, Dongyan Xu
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

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

Xuxian Jiang, Xinyuan Wang
Department of Information and Software Engineering
George Mason University
{xjiang, xwangc}@ise.gmu.edu

Dongyan Xu
Department of Computer Science
Purdue University
dxu@cs.purdue.edu

ABSTRACT

An alarming trend in malware attacks 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 native, semantic view of the host which is enjoyed by the "in the box" approach, thus leading to 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 systematically reconstruct internal semantic views (e.g., files, processes, and kernel modules) of a VM from the outside in a non-intrusive manner. 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. With the semantic gap bridged, we identify two unique malware detection capabilities: (1) *view comparison-based malware detection* and its demonstration in rootkit detection and (2) *"out-of-the-box" deployment of host-based anti-malware software* with improved detection accuracy and tamper-resistance. We have implemented a proof-of-concept prototype on both Linux and Windows platforms and our experimental results with real-world malware, including elusive kernel-level rootkits, demonstrate its practicality and effectiveness.

Categories and Subject Descriptors D.4.6 [**Operating System**]: Security and protection – Invasive software

General Terms Security

Keywords Malware Detection, Rootkits, Virtual Machines

1. INTRODUCTION

Internet malware (e.g., rootkits and bots) is getting increasingly stealthy and elusive: they strive not only to hide their presence from detection facilities in the compromised system, but also to detect and subvert existing anti-malware software. A detailed analysis of an Agobot variant [1] has revealed that the malware contains malicious logic to detect and remove more than 105 anti-virus processes in the victim machine.

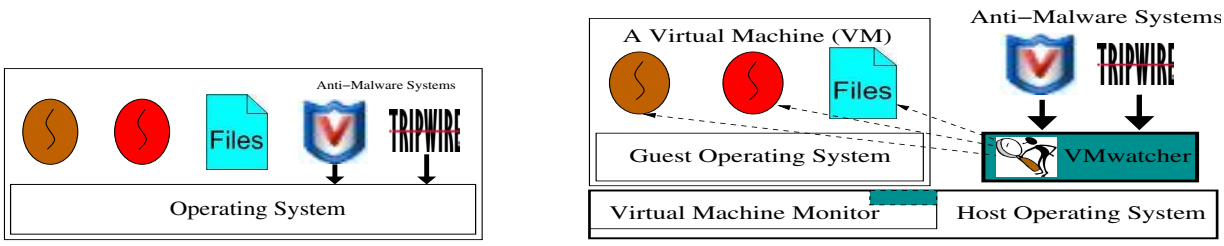
The threats above are partly attributed to a fundamental limitation on the defensive side: Most host-based anti-malware systems are installed and executed inside the very hosts that they are monitoring and protecting (Figure 1(a)). Although such "in the box" deployment will provide an anti-malware system 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 [32, 34, 37] that advocate placing the intrusion detection facilities outside of the (virtual) machine being monitored. Based on virtual machine technologies [17, 26, 31], such an "out of the box" approach significantly improves the tamper-resistance of intrusion detection facilities. A virtual machine (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" [29] 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 it difficult for "out of the box" malware detection. In other words, the "out of the box" approach gains tamper resistance at the cost of losing the native, semantic view of the host enjoyed by the "in the box" approach.

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) [34] methodology in a non-intrusive manner so that it can inspect the low-level VM states without perturbing the VM's execution. Furthermore, a new technique called *guest view casting* is developed to systematically re-constructing the VM's internal semantic view (e.g., files, directories, processes, and kernel-level modules) for out-of-the-box malware detection. The new technique is based on the key observation that the guest OS of a VM provides all necessary semantic definitions of guess data structures and functions to

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.



(a) The traditional "in the box" approach

(b) The VMwatcher approach

Figure 1: Malware detection in the traditional "in the box" approach and in our VMwatcher approach

construct the VM's semantic view. As such, we can cast them on the VMM-level observations and externally reconstruct the semantic view of the target VM (Figure 1(b)).

VMwatcher provides new capabilities of detecting stealthy malware that are previously difficult or impossible to achieve. In this paper, we identify and demonstrate two 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 and (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-virus software developed for one platform can be readily used for another platform.

We have implemented a VMwatcher prototype on both Linux and Windows platforms and evaluated it with a collection of real-world malware instances (e.g., kernel and user-level rootkits). Experimental results with these elusive rootkits demonstrate VMwatcher's unique capability of enabling view comparison-based malware detection. Our VMwatcher prototype also supports out-of-the-box deployment of a variety of off-the-shelf anti-malware software such as Symantec AntiVirus [12] and Microsoft Windows Defender [7].

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 then present evaluation results in Section 4 and discuss possible limitations in Section 5. Finally, 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 tamper-resistance by moving malware monitoring facilities out of the VM being monitored. It is based on two key enabling techniques: (1) non-intrusive VM introspection for the procurement of low-level (VMM-level) VM state without relying on any facility inside the VM (Section 2.2.1) and (2) guest view casting for the 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 VM being monitored. This will prevent VMwatcher from affecting the normal execution of the VM and causing adverse side effects (e.g., system inconsistency [37]) in the VM. This goal is realized by our technique for non-intrusive inspection and analysis of low-level VM states. 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 in enabling the new stealthy malware detection capabilities. Specifically, the goal is realized by our guest view casting technique for external reconstruction of VM semantic view. Based on the reconstructed view, file or memory scanning operations of anti-malware systems can be performed as if they were inside the VM [1].

Third, VMwatcher should be generic and applicable to a wide range of existing VMMs. Currently there exist two mainstream virtualization approaches: full virtualization and para-virtualization. Full virtualization (as in VMware [17] and QEMU [27]) transparently supports legacy OSes without modifying the guest OS implementation; while para-virtualization (as in Xen [26] and User-Mode Linux [31]) is less transparent as it needs to modify the source code of guest OSes. VMwatcher supports 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 chosen to implement a VM, 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 (enabled by *ptrace* [31]), User-Mode Linux (UML) preserves much of the semantic information (e.g., processes) and thus leads to a much narrower semantic gap than in VMware, Xen, and QEMU.

Assumption on trusted VMM In this paper, we assume a trustworthy VMM model: A malware instance can 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 model is based on the observation that the code base of a VMM is much smaller and more stable than the code in the legacy OSes. Further, it only needs to provide a rather limited interface (that can be further examined and hardened) to untrusted VMs in the form of abstracting underlying physical resources. Note that this assumption is consistent with that of many other VMM-based security research efforts [32, 33, 34, 37, 42]. We will discuss possible attacks (e.g., VM fingerprinting) in Section 5.

[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, cannot be easily captured from low-level VMM observations.

2.2 Enabling Techniques

2.2.1 Non-Intrusive Virtual Machine Introspection

VMwatcher follows the VM introspection methodology to procure low-level VM states externally. For open-source VMMs such as Xen, QEMU, and UML, we develop non-intrusive VM introspection extensions to obtain full VM state including the VM's registers, memory, and disk. To achieve non-intrusiveness, we follow the principle of passive observation with no active influence on the VM – this is important as an improper influence would introduce undesirable consequences such as inconsistency in the VM's system state or perturbance in the VM's execution.

For close-source VMMs such as VMware, we have a more limited access to VMM-level observations. For example, we are not able to read virtual machine registers (e.g., the control register CR3) or monitor virtual interrupts using the off-the-shelf VMware. Without the VMM's source code, VMwatcher has to rely on whatever low-level VM state abstraction exposed by the VMM. In the case of VMware, the limited VM state view includes the virtual disk and physical memory. Fortunately, these limited VM observations still allow for the reconstruction of semantic views that are sufficiently rich for important malware detection operations such as file and memory scanning. Details of our non-intrusive VM introspection technique will be presented in Section 3.

2.2.2 Guest View Casting

Given a VMM-level VM state, our second technique, guest view casting, will externally reconstruct the semantic-level view of the VM thus bridging the semantic gap. We observe that the guest OS data structure definitions (e.g., files and directories) and function semantics (e.g., that of file system drivers) can be used as "templates" to interpret low-level VM states. As such, we can cast these guest data structures and function semantics on the VMM-level VM observations so that the VM's semantic view can be recreated externally. 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 individual running process with its attributes such as PID and process 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 are presented to an anti-malware system in exactly the same way as inside the VM. For example, Tripwire [38] assumes a standard UNIX-like file system layout and calculates the checksums of files and directories to identify possible changes; McAfee VirusScan examines local 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.

Finally, we point out that guest view casting is performed outside of the target VM. Based on the trustworthy VMM model (Section 2.1), any software running inside the VM is not able to tamper with the external reconstruction of VM semantic view. Moreover, the fact that VMM-level VM states are procured through non-intrusive VM introspection implies that any malware instance inside a target VM is not able to infer or influence the semantic view reconstruction activities of VMwatcher.

2.3 New Malware Detection Capabilities

VMwatcher provides the technical basis for a number of new malware detection capabilities. The first capability is view comparison-based detection of self-hiding malware. We have seen an increasing number of elusive malware instances that actively hide themselves as well as related files or processes by subverting anti-virus processes running inside a system. With view comparison, we can corroborate an internal view (generated from inside the VM) with an external view (generated from outside the VM) of the same objects of interest and detect the existence of hidden malware based on the discrepancy exposed. We note that view comparison can be based on either the full semantic views of a VM, or 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 about a particular 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 show one such example in Section 4.2.

3. IMPLEMENTATION

We have implemented a prototype of VMwatcher, which supports four existing VMMs: VMware, QEMU, Xen, and UML. The same design and implementation methodologies could also be applied to other VMMs such as KVM [5] and VirtualBox [16]. In addition, VMwatcher 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 the implementation details, with a focus on VMM-level VM state procurement and semantic-level VM view reconstruction.

3.1 VMM-Level State Procurement

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., machine registers)	×	√	√	√
VM-related low-level events (e.g., interrupts/traps)	×	√	√	√

Table 1: VMM-level VM state observations

As mentioned in Section 2.1, VMwatcher is designed to be generically applicable to various VMMs. As a result, our prototype is

based on VMM-level VM state abstractions commonly supported by these VMMs. Table 1 lists the VMM-level VM state observations offered by the four VMMs supported: the open-source VMMs – QEMU, Xen, and UML – allow full access to low-level VM states and events; while the close-source VMware only exposes the raw disk blocks and raw memory pages allocated to a VM.

We focus our presentation on the procurement of a VM's raw disk and memory states. More specifically, we need to access a VM's raw disk and memory while they are being modified by a running VM. To ensure state consistency, a VMM usually grants an exclusive access (e.g., with a *write* lock) to the virtualized resources (e.g., memory or disk) to a VM. As a result, it could prevent any external process from accessing them. Specifically, the file lock in Windows imposed by a running VMware-based VM instance is *mandatory*, which means that any other 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 [18]) 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 on the Windows platform 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* [19] by default, which means 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[28] 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 for inspection and procurement. However, Xen and QEMU (without KQEMU support) do not create the corresponding memory file. As such, we need to extend them to export a VM's physical memory pages. Fortunately, the open-source nature of Xen and QEMU facilitates our solution. In our prototype, VMwatcher takes advantage of the *libxc* library [22] to access the memory of a Xen-based VM (or DomU) by mapping its physical memory with the API named *xc_map_foreign_range()* to its address space and then reading the content through the mapped memory. Similarly, we build our own library for QEMU, which essentially allows for external access of VMwatcher to the allocated physical memory pages for a QEMU-based VM.

3.2 Semantic View Reconstruction

Based on the VMM-level VM raw 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 the two main virtualized resources.

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 as well as 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 our VMwatcher prototype, we have written Windows device drivers to interpret Linux file systems (*ext2/ext3* root file systems).

Memory state reconstruction: It is a more challenging task to reconstruct the semantic view of volatile VM memory. Similar to the disk, we are able to procure the *physical* memory pages allocated to a VM by the VMM. However, 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 the Linux platform with the current 32-bit architecture (which implies the 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, the offset 0 in the memory file will correspond to the current memory address $0xC0000000$ inside the VM. Next, we utilize the exported symbol information [2], and apply guest view casting to identify and reconstruct those guest data structures of interest. Figure 2 shows how guest view casting can be applied to reconstruct the volatile kernel memory state of a Linux-based VM. 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 containing core kernel instructions or instructions in the loadable kernel modules. It is worth mentioning that when accessing a user-level memory address ($< 3G$), it is usually referring to a virtual memory address specific to a particular process running inside 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 accessed through the low-level VMM observations.

We note that existing hardware has the capability of automating the process of traversing the page table for the address translation. However, it has the implicit assumption that the running process has the same page table base (CR3) as the memory address to be accessed. As a result, our prototype needs to externally identify and walk through the page table of an internal process to obtain the

[2]For some commercial OSes such as Windows, which may not provide the locations of these important symbols, VMwatcher will perform a full scan on the raw memory and identify them by looking for certain "signatures" [24] that are unique to kernel-level data structures of interest. For example, we have used so far $0x03001b0000000000$ to identify potential process instances in the Windows XP raw memory file.

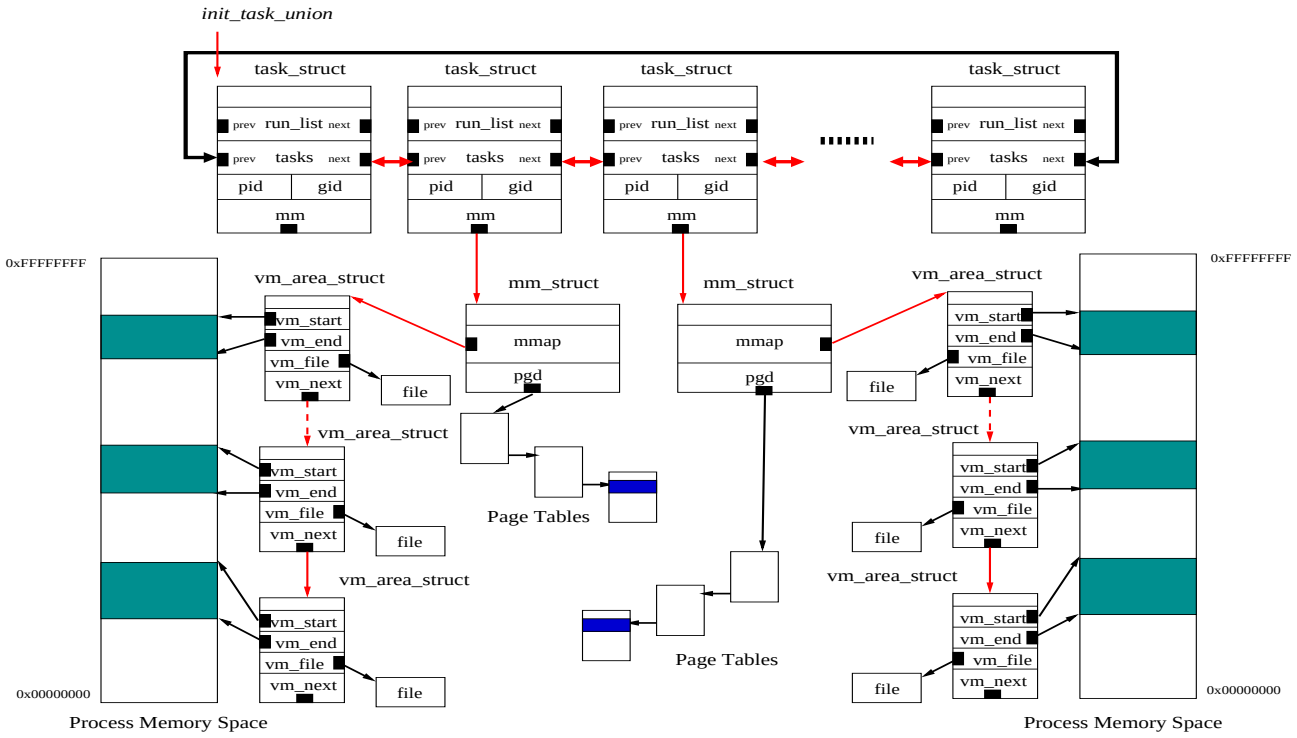


Figure 2: Guest view casting for volatile VM memory state (in Linux)

corresponding physical memory address and read its content for inspection. The corresponding code is illustrated below in function *vmwatcher_vir_mem_read32*, where *addr* is the virtual memory address to be queried; *task* points to the process control block (assuming the *task_struct* data structure in Figure 2) of an internal process of interest; *pde* and *pte* refer to a page directory entry and a page table entry associated with the internal process, respectively; and *vmwatcher_phy_mem_read32* reads the actual physical memory content with the given physical memory address from VMM-based observations.

```
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 above description is in the context of Linux, our guest view casting-based semantic view reconstruction provides a generic, systematic methodology that can be applied to various VMM platforms and operating systems. During the prototype implementation, we have evaluated how different operating systems, service patches, and system configurations impact the casting of VM states and events. For example, operating systems may have different memory layouts (e.g., the Windows OS has a 2G/2G memory split between user and kernel space), 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. 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. However, the guest view casting methodology remains effective despite these differences, as shown by our evaluation in Section 4.

4. EVALUATION

We evaluate our prototype to demonstrate the two new malware detection capabilities (Section 2.3) enabled by VMwatcher. In particular, we show: (1) how the view comparison-based scheme effectively detects one of the most stealthy malware – rootkits (Section 4.1) and (2) how VMwatcher enables "out of the box" execution of legacy anti-malware systems (Section 4.2). Finally, we present performance measurement results in Section 4.3.

4.1 View Comparison-based Malware Detection

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

Experiment I – view comparison on volatile states The first experiment involves a Windows kernel-level FU rootkit [3]. 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*

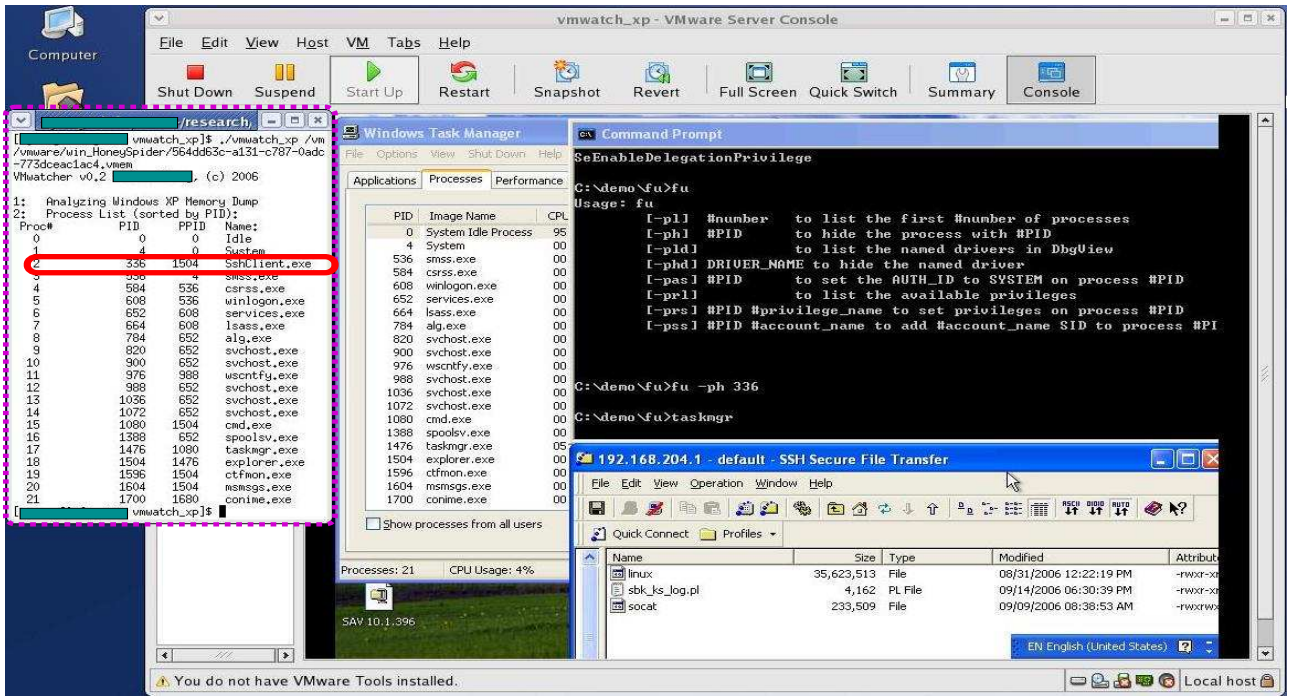
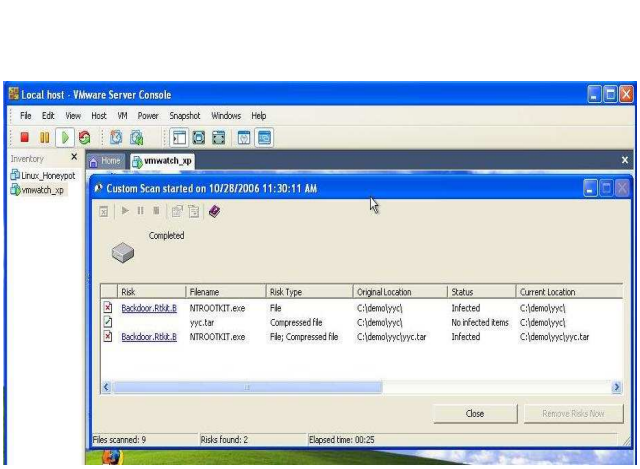
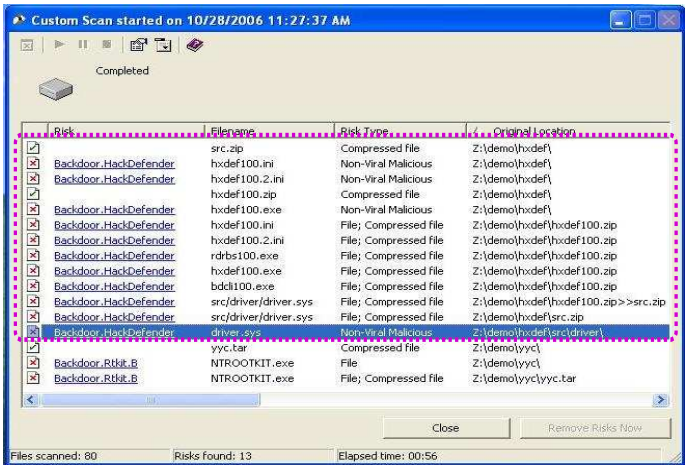


Figure 3: A VMware-based Windows XP VM infected by the FU rootkit



(a) Result of running Symantec AntiVirus from inside



(b) Result of running Symantec AntiVirus from outside

Figure 4: A demonstration of different views obtained from inside and outside of a hxdef-infected VM

of the VM while the foreground screen (encapsulated with a dashed box) on the left shows the VMwatcher-based *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.

By comparison, the hidden process is exposed by the VMwatcher-based external view: The small box with solid lines inside the foreground screen highlights the *SshClient.exe* 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 [9] show that the same FU rootkit has been actively used to hide the presence of advanced stealthy bots.

Experiment II – view comparison on persistent states In this experiment, we prepare a VMware-based Windows XP VM that

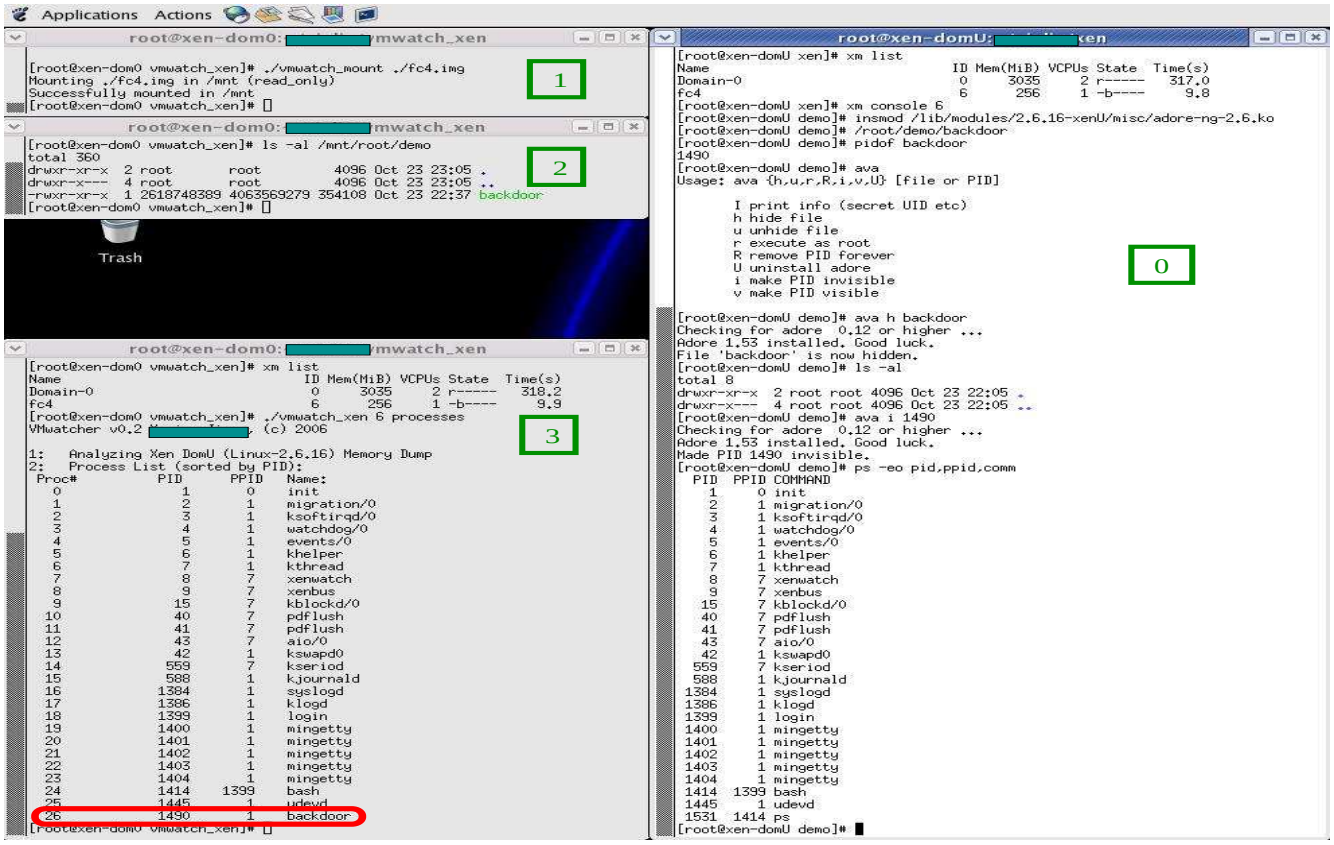


Figure 5: A Xen-based Fedora Core 4 VM infected by the adore-ng rootkit

contains the files of two rootkits, Hacker Defender (or hxdef) [4] and NTRootkit [8], in the $c:\backslash demo$ 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*.

Experiment III – view comparison on *both* volatile and persistent states We describe our experiment with the *adore-ng* [13] rootkit – an advanced Linux kernel rootkit that will directly replace certain kernel-level function pointers to hide files and processes.

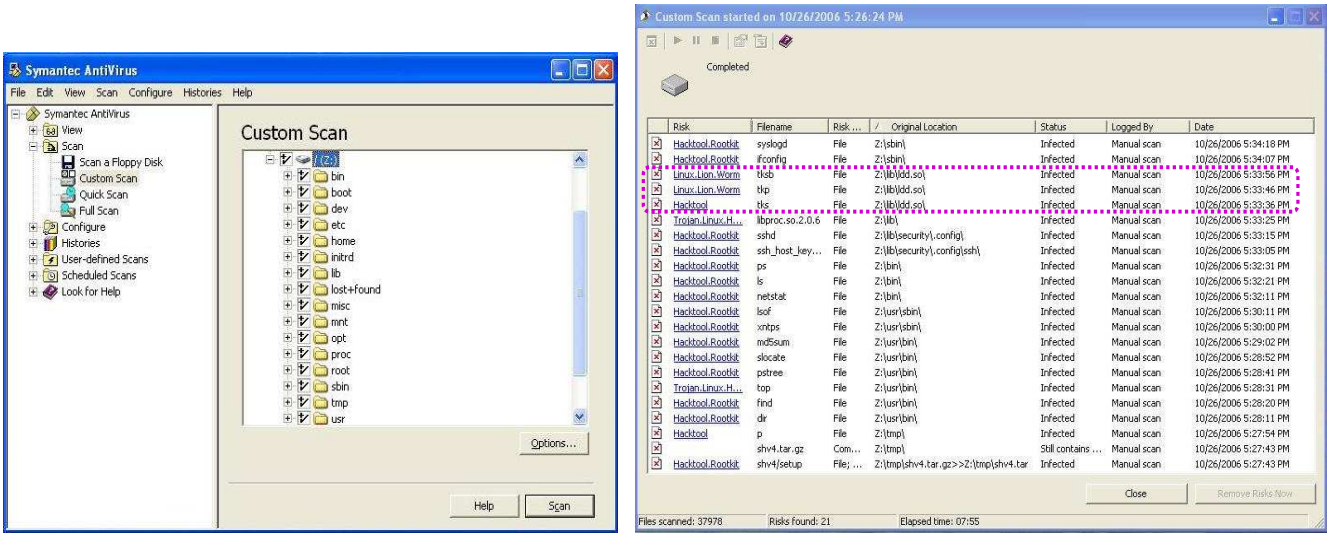
Figure 5 is a screenshot showing an *adore-ng* infection against a Xen-based Fedora Core 4 VM. Within the figure, there are four numbered *xterm* windows. The *xterm* window with the number 0 on the right shows the inside of the VM, where the *adore-ng* kernel-level module (LKM) is first loaded (*insmod /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 whose PID is 1490 (*ava i 1490*). As revealed in the same xterm window, the outputs from the commands *ls* and *ps* are already manipulated to *conceal* the existences 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* with 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 existence of a rootkit (even if the hidden file and process have unsuspected names).

4.2 "Out-of-the-Box" Malware Detection

By externally reconstructing semantic views of VMs, VMwatcher also supports "out-of-the-box" execution of a number of off-the-shelf anti-malware systems and naturally brings up the new capability of cross-platform malware detection. We have successfully experimented with 11 real-world anti-virus software systems, which are shown in Table 2. For each experimented anti-virus software, Table 2 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 VM honeypot.



(a) A screenshot of the Symantec AntiVirus software before launching its scanning

(b) A screenshot of the Symantec AntiVirus software after completing its scanning

Figure 6: External inspection of a honeypot using the Symantec AntiVirus software (version 10.1.0.396)

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 2: A list of real-world anti-virus software we have experimented with VMwatcher.

Experiment IV – cross-platform malware detection The Linux honeypot is a VMware-based Red Hat 7.2 system 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* launching the scanning and one *after* completing the scanning. 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. Particularly, among those infected files, there is a rootkit named SHv4 [47], which replaces a number of system-wide utility commands (e.g., ps, ls, ifconfig, netstat, and syslogd etc.) with its own. We also notice that there is a Lion worm infection in the report (highlighted in the dashed box of Figure 6(b)), which we believe is misclassified. The two identified Lion-infected files are *tksb* and *tkp* under the 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 [20] to gain system access. After that, he exploited the local ptrace kernel vulnerability [21] to escalate his privilege to system root before installing the SHv4 rootkit.

For comparison, we also run Microsoft Windows Defender (version 1.1.1592.0) in the host domain to detect possible malware installations in this compromised VM and the result, interestingly, shows *no* malware infection. It seems that the current Microsoft Windows Defender is developed only for malware on Windows platforms while the Symantec AntiVirus software is capable of detecting malware on both Windows and Linux platforms.

4.3 Performance

In this section, we present the performance measurement results. We first note that VMwatcher is operated outside of a VM. As a result, it will not affect the normal run of a VM even when it is being examined. In the following, We present two sets of measurement results.

The first set of experiments is to compare the internal scanning time and the external scanning time on a set of VM systems. In par-

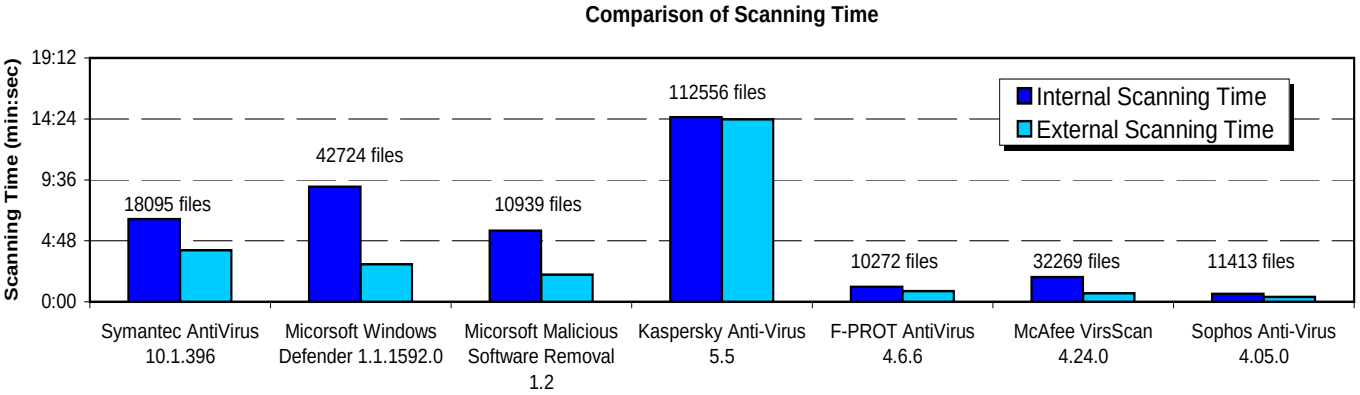


Figure 7: A comparison of internal scanning time and external scanning time

ticular, we choose 7 different anti-virus software systems and each system performs an external scan and an internal scan on a particular VM system: (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 the 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 are assigned to look into a Red Hat 7.0 VM (128MB memory and 512MB disk) that is running inside a UML VMM. The host OS is Red Hat Enterprise Linux 4 with 2GB memory and 135GB disk. The results plus the total number of scanned files are shown in Figure 7. It is interesting to notice that an internal examination always takes longer scanning 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.

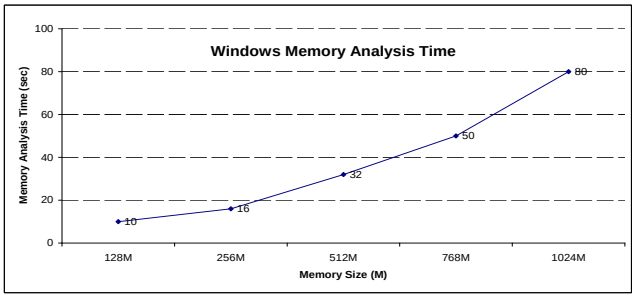


Figure 8: Memory analysis time

The second set of experiments calculates the time needed to analyze a live raw VM memory. Note that in the current 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-related semantic view reconstruction. Figure 8 shows the analysis 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 linearly with the size of available memory allocated to a VM. Our results show that with the availability of Linux symbols, a raw memory analysis session can be finished within just 0.5 second, regardless of the allocated memory size of the VM.

5. DISCUSSION

VMwatcher assumes a trustworthy VMM layer to isolate untrusted processes inside a monitored VM from affecting VMwatcher. This assumption is needed and reasonable (some parallel efforts to build trustworthy VMMs will be described in Section 6) because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. In the following, we discuss some possible attacks against VMwatcher.

Guest view subversion attack This attack is based on the observation that VMwatcher needs to correctly cast guest views for the interpretation and understanding of guest VM states. As such, an attacker can intentionally introduce a subverted guest function, which is different from the one casted for semantic view reconstruction by VMwatcher. As an 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 actually linking them into the all-tasks list. Note that without the knowledge of the subverted scheduler, VMwatcher is not able to accurately identify all running processes.

Although it is challenging to understand the details of subverted guest functions, the subversion behavior itself could be detected. Considering the same example, the subversion of the original scheduler code will essentially modify the text segment of the original Linux kernel. A simple hash calculation (e.g,. MD5) can immediately lead to its detection. As such, to counter this type of attacks, VMwatcher can be extended to validate the integrity of these guest functions as well as other critical kernel objects (e.g., *sys_call_table* and IDT). Moreover, we can leverage recent research efforts such as Copilot [51] and the related specification-based integrity checking [50] to detect these subversion attacks.

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 transparently to the external anti-virus processes.

VM fingerprinting Finally, we note that the virtualized environment could potentially be fingerprinted and detected [41, 52] by attackers. In fact, a number of recent malware systems are able to check whether they are running inside a VM and if so, choose to exhibit different behavior [1]. As a counter-measure, we can improve the fidelity of VM implementation (e.g., as proposed in [43, 45]) to thwart some of the VM detection schemes. Meanwhile, from another perspective, as virtualization continues to gain popularity, the concern over VM detection may become less significant because most malware would become VMM-agnostic once again as VMs could be attractive targets for attackers as well.

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 [34, 37, 44], analyze intrusions [32, 42], diagnose system problems [40, 57], isolate services [30, 46], and implement honeypots [14, 23, 35]. 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 [34], which is pioneered by the Livewire system [34]. 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 [34].

IntroVirt [37] 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 [37]. 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 [39, 53, 58]. King et al. [39] proposes the notion of VM-based rootkit (VMBR) which can be dynamically inserted underneath an existing OS. Rutkowska et al. [53] 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 [58] rootkit – independently confirms this significant threat. We point out that these emerging threats can be mitigated or even defeated by recent research efforts on secure booting [25] and secure VMMs such as sHype [54] and TRANGO [15]. 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 [33, 49, 50, 51]. Copilot [51] 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 [50] 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 [49] 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 for malware detection.

Detecting malware with cross-view comparison The notion of view comparison-based analysis is initially proposed by Wang et al. [55] 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 procurement and semantic view reconstruction without losing any malware information. A number of recent rootkit detection systems such as RootkitRevealer [11] and BlackLight [2] 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 [6, 7, 12, 38] and the network-based IDS [10, 48, 56]. 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.

7. CONCLUSION

We have presented VMwatcher, a novel VMM-based approach that enables "out-of-the-box" malware detection 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 on both Linux and Windows platforms demonstrates its practicality and effectiveness. In particular, our experiments with real-world self-hiding rootkits further demonstrate the power of the new malware detection capabilities enabled by VMwatcher.

8. REFERENCES

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

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

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

[4] hxdef. *http://hxdef.czweb.org*.

[5] KVM: Kernel-based Virtual Machine. *http://kvm.sourceforge.net/*.

[6] McAfee VirusScan. *http://www.mcafee.com/us/enterprise/products/anti_virus/*.

[7] Microsoft Windows Defender. *http://www.microsoft.com/athome/security/spyware/software/default.mspx*.

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

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

[10] Snort. *http://www.snort.org*.

[11] RootkitRevealer. *http://www.sysinternals.com/*.

[12] Symantec AntiVirus. *http://www.symantec.com/home_homeoffice/products/overview.jsp?pcid=is&pvid=nav2007*.

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

[14] The Honeynet Project. *http://www.honeynet.org*.

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

[16] VirtualBox. *http://www.virtualbox.org/*.

[17] VMware. *http://www.vmware.com/*.

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

[19] Wikipedia: File Locking. *http://en.wikipedia.org/wiki/File_locking*.

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

[21] Linux Kernel Ptrace Privilege Escalation Vulnerability. *http://www.secunia.com/advisories/8337/*, Mar. 2003.

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

[23] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. *Proc. of the 14th USENIX Security Symposium*, Aug. 2005.

[24] bugcheck. GREPEXEC: Grepping Executive Objects from Pool Memory. *http://www.uninformed.org/?v=4&a=2&t=sumry*, June 2006.

[25] W. A. Arbaugh, D. J. Farbert, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. *Proc. of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. N. A. Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proc. of the 2003 SOSP*, Oct. 2003.

[27] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track)*, July 2005.

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

[29] Peter M. Chen and Brian D. Noble. When Virtual is Better Than Real. *HotOS VIII, Schoss Elmau, Germany*, May 2001.

[30] E. Bryant, J. Early, R. Gopalakrishna, G. Roth, E. H. Spafford, K. Watson, P. Williams, and S. Yost. Poly2 Paradigm: A Secure Network Service Architecture. *Proc. of the 19th Annual Computer Security Applications Conference*, Dec. 2003.

[31] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[32] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *Proc. of USENIX OSDI 2002*, Dec. 2002.

[33] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *Proc. of the 2003 SOSP*, Oct. 2003.

[34] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proc. of the 2003 Network and Distributed System Security Symposium*, Feb. 2003.

[35] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proc. of the 13th USENIX Security Symposium*, Aug. 2004.

[36] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *Proc. of the 8th International Symposium on Recent Advances in Intrusion Detection*, September 2005.

[37] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. *Proc. of the 2005 SOSP*, Oct. 2005.

[38] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. *In Systems Administration, Networking and Security Conference III, USENIX*, 1994.

[39] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[40] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *Proc. of the 2005 Annual USENIX Technical Conference*, Apr. 2005.

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

[42] T. Koju, S. Takada, and N. Doi. An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, June 2005.

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

[44] K. Kourai and S. Chiba. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, June 2005.

[45] T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. *http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf*, 2006.

[46] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, Sept. 2000.

[47] J. V. Miller. SHV4 Rootkit Analysis. *https://tms.symantec.com/members/AnalystReports/030929-Analysis-SHV4Rootkit.pdf*.

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

[49] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. *Proc. of the 12th USENIX Security Symposium*, Aug. 2003.

[50] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. *Proc. of the 15th USENIX Security Symposium*, Aug. 2006.

[51] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *Proc. of the 13th USENIX Security Symposium*, Aug. 2004.

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

[53] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. *Blackhat 2006*, Aug. 2006.

[54] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. *IBM Research Report RC23511*, Feb. 2005.

[55] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. *Proc. of the 2005 International Conference on Dependable Systems and Networks*, June 2005.

[56] N. Weaver, V. Paxson, and J. Gonzalez. The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. *Proc. of the FPGA 2007*, Feb. 2007.

[57] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. *Proc. of USENIX OSDI 2004*, Dec. 2004.

[58] D. D. Zovi. Hardware Virtualization Based Rootkits. *Blackhat 2006*.