VMDriver: A Driver-based Monitoring Mechanism for Virtualization

Guofu Xiang†, Hai Jin†, Deqing Zou†, Xinwen Zhang‡, Sha Wen†, Feng Zhao†
†Huazhong University of Science and Technology, Wuhan, 430074, China
‡Samsung Information Systems America, San Jose, CA 95134, USA
hjin@hust.edu.cn xinwen.z@samsung.com

***Abstract*—Monitoring virtual machine (VM) is an essential function for virtualized platforms. Existing solutions are either coarse-grained–monitoring in granularity of VM level, or not general–only support specific monitoring functions for particular guest operating system (OS). Thus they do not satisfy the monitoring requirement in large-scale server cluster such as data center and public cloud platform, where each physical platform runs hundreds of VMs with different guest OSes. As a result of this reason, we propose VMDriver, a general and fine-grained approach for virtualization monitoring. The novel design of VMDriver is the separation of event interception point in VMM level and rich guest OS semantic reconstruction in management domain. With this design, variant monitoring drivers in management domain can mask the differences of guest OSes. We implement VMDriver on Xen and our experimental study shows that it introduces very small performance overhead. We demonstrate its generality by inspecting four aspects information about the target virtual machines with different guest OSes. The unified interface of VMDriver brings convenience to develop complex monitoring tools for distributed virtualization environment.**

Keywords-Virtualization, VM Monitoring; Generality; Driver-based Monitroing; Event Interception; Semantic Reconstruction

I. INTRODUCTION

Virtualization has been widely used in server cluster and data center to consolidate diverse services on a few servers and multiplex underlying hardware resource [1][2][3]. Monitoring is an intrinsic function for reliability and security of virtualized computing environment. Many management functions and tools are built on accurate and efficient monitoring mechanism to check system state and schedule resource allocation into VM in real time manner. For example, more memory can be allocated to one VM when a service running inside it receives more requests than usual. For another example, when one VM is corrupted by attacker, security mechanism based on monitoring technology may decide to terminate this VM in order to prevent the corruption of others.

Typical virtualized platform has one management VM (MVM) (e.g., `Dom0` on Xen) which is engaged in monitoring one or more target VMs (TVMs [1]) (e.g., `DomU` on Xen). From a high level view, existing monitoring mechanism can be classified into two types: performance monitoring and security monitoring. Performance monitoring usually takes a complete TVM as monitoring granularity without considering guest OS in it. Furthermore, it focuses on performance aspect such as CPU usage, memory size, and network bandwidth. Take Xen platform [4][5][6] as example, tools like `xentop`, `xentrace`, `xenperf` inspect the running state and resource consumption of all TVMs. The platform administrator can adjust resource allocation for individual TVMs according to information obtained from these monitoring tools.

However, performance monitoring on VM level does not give detailed information of internal status for guest OS, and fine-grained monitoring is critical for some monitoring purposes such as security. Therefore many fine-grained monitoring tools have been proposed for virtualized platform, for example intrusion detection [7], honeypot [8], malware detection [9][10], malware analysis [11][12][13], and security monitoring architecture [14]. All kinds of these monitoring functions are in the virtual machine monitor (VMM), and thus do not affect the runtime behavior of the TVM. As they focus on specific security purposes, these tools usually do not satisfy the monitoring task in distributed and large-scale virtual computing environment such as data center and cloud computing platform (e.g., Amazon EC2), where variant OSes (different types or versions) may run in individual TVMs. For example, OS-dependent semantic reconstruction is used in many existing security monitoring mechanisms. When a new VM with different OS starts locally or migrates from another computing node, the monitoring tool may become invalid.

Aiming to build *fine-grained and at the same time general monitoring mechanism* for virtualized platform, we propose VMDriver, a driver-based monitoring architecture which adopts the mechanism analogous to the device driver concept in Linux. VMDriver achieves these objectives with two novel design strategies. First, guest OS event interception and semantic reconstruction are separated into two parts in VMM and MVM respectively. Event interception captures the variation of system state underneath the TVM, and semantic reconstruction shields the diversity of guest OSes in these TVMs. Secondly, semantic reconstruction in VMDriver is handled by a set of monitoring drivers in MVM. Similar to Linux device driver which shields the diversity of underlying hardware device for user-level application, the monitoring

[1]TVM, guest VM, and VM are interchangeable terms in this paper.

driver in VMDriver encapsulates related information about a TVM corresponding to the specific guest OS. When an event happens in the TVM, the event sensor residing in VMM intercepts it and reports to the corresponding monitoring driver in MVM. The monitoring driver reconstructs the semantic information under the context of guest OS. With this, a set of monitoring drivers running in MVM mask the differences of multiple TVMs and expose the same interface to other management tools and platform administrators. VMDriver enables several novel benefits for general monitoring purpose. First, multiple monitoring tools can be deployed on a single physical platform for TVMs with variant OSes, which is critical for the business success of large data center and cloud computing platform. Secondly, VMDriver enables existing monitoring tools to be useful for multiple guest OSes, instead of the specific one. For example, a single malware detection tool can detect malicious behavior both in Linux and Windows with the same interface provided by VMDriver.

We have implemented VMDriver on Xen. Event interception function resides in Xen and diverse monitoring drivers are implemented as kernel modules in `Dom0`. Experimental study confirms that VMDriver is effective on monitoring different OSes in VMs. In addition, it provides general interface for other monitoring tools such as malware detection, and has small performance overhead.

The rest of this paper is organized as follows. Section II introduces the related work on performance monitoring and security monitoring. In Section III, we describe the problem in server cluster and data center as the application scenario, and present the overview and design requirement of VMDriver. Section IV presents the implementation detail of VMDriver on Xen, and section V shows the experiment and evaluation. After that, we discuss the advantage of VMDriver and point out its limitation at the same time. At last, we conclude this paper in Section VI.

II. Related Work

A. Performance Monitoring

Performance monitoring tools are usually distributed with virtualization system software such as Xen [4], VMware [15], and Virtual PC [16]. They are used to manage and control TVMs by system administrator. For example, VMware provides graphical management for controlling all TVMs and software development kit (SDK) for developers. Beside graphical interface, Xen also offers command mode of monitoring tools. The administrator can use `xm` to send the requests and daemon `xend` responses these requests. `xentop` displays real-time information about all TVMs, and `xentrace` captures trace buffer data from Xen. All these performance monitoring tools are coarse-grained as they take a complete VM as monitoring granularity. Because of this, they are independent from guest OS and do not have the generality problem.

B. Security Monitoring

With the unique position on a virtualized platform, VMM has been leveraged for fine-grained monitoring for TVMs. By observing TVM events at VMM level, VMM-based approach does not affect the runtime behavior of TVM and is resistant to malicious modification from TVM. Observing system state outside a monitored system is called virtual machine introspection (VMI), which is firstly proposed by Garfinkel and Rosenblum in Livewire [7]. In this way, intrusion detection is built with the merits of both high resistance and excellent visibility. VMscope views a VMM-based honeypot's internal events from outside [8]. Jones et al. propose a VMM-based detection and identification service for hidden processes [9]. VMWatcher enables "out-of-the-box" malware detection by addressing semantic gap between observed events at VMM level and guest OS context in the TVM. Moreover, it implements strong tamper-resistance by moving anti-malware facilities out of VM [10]. Ether is a transparent and external approach for malware analysis [11], which implements monitoring instruction execution, memory write, system call execution, and limited scope of a chosen process by hardware virtualization extension. K-Tracer dynamically analyzes Windows kernel-level code and extracts malicious behavior from rootkits [12]. Rkprofiler is a sandbox-based malware tracking system for dynamic monitoring and analyzing the behavior of Windows kernel malware [13]. Lares is an active architecture for security monitoring using virtualization [14]. In this framework, monitoring hooks are installed in an unreliable TVM, which is protected by VMM. In 2008, VMware released the virtualization-based library - VMsafe [17]. It provides a set of API for other security manufacturers to improve the security of virtualization architecture. Xenaccess [18] is an open source monitoring library on Xen, which provides a convenient interface for programmers by encapsulating `libxc` and `libblktap` supplied by Xen.

Most of the above work focus on specific function of virtualization monitoring and target for specific type of TVM. Xenaccess is designed for general memory and disk introspection, but it can be evaded by [19]. In data center and cloud platform, TVMs running on single physical platform differ with each other. When a new TVM is launched on or migrated to a physical platform, these secure tools may fail to provide effective monitoring. Thus, they can not fit the requirement of heterogeneous TVMs. The design objective of this paper is the generality of monitoring TVMs with various guest OSes and a unified interface for high-level monitoring tools.

III. VMDriver Overview

A. Problem Statement

Server cluster and data center are armored with powerful processing nodes which are usually virtualized and have

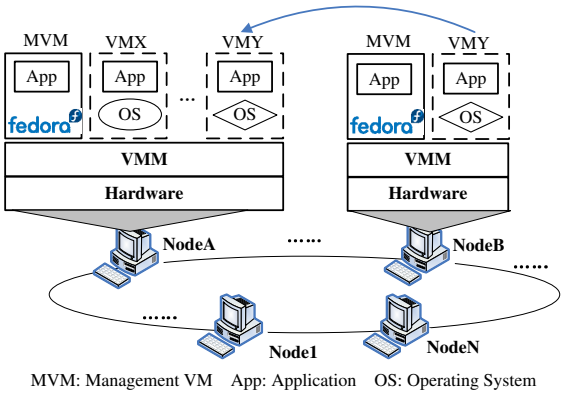


Figure 1. Virtualized platform nodes in data center

multiple TVMs running at the same time. Typically, the MVM inspects all TVMs running on the same node.

Consider two nodes, `NodeA` and `NodeB`, as shown in Figure 1. Initially we assume `NodeA` has only the MVM and one TVM (`VMX`). In order to consolidate service or solve hardware fault, another TVM (`VMY`) running on `NodeB` is migrated to `NodeA`. The TVMs on `NodeA` and `NodeB` vary with time, and they can migrate between these physical nodes according to resource scheduling strategy. It is essential to monitor the states of all TVMs in time such that the platform administrator or service provider can take corresponding action according to the inspection result. The dynamic execution environment of single node makes this task challenging in real world: it is hard to inspect the states of all TVMs with current monitoring mechanism. Either they are not fine-grained enough to capture the internal state of guest OS, or they focus on specific function and OS environment thus lack generality. The problem is salient in large-scale server cluster and data center, where a single physical node may have hundreds of TVMs concurrently running with different type and version of guest OSes.

For fine-grained monitoring, the MVM on a physical node can receive internal state information of a TVM with the help of VMM. Due to the position of VMM, the obtained state information is at binary level, such as registers and memory pages. Nevertheless, security tools execute policy at system level (e.g., process, file). For this reason, it is necessary to translate binary information to system information, which is named semantic reconstruction. Figure 2 gives an example of system call interception and semantic reconstruction at the VMM layer, which is expressed in a C-like pseudo-code. Whenever a system call happens in the TVM, the VMM intercepts it with function `intercept_syscall` and returns current CPU registers (`regs_info`) (line 18 in Figure 2). After that, semantic reconstruction is implemented by function `syscall_handler`, and binary information is translated to system semantic (line 19 in

```
1  typedef struct cpu_regs
2  {
3     uint32_t eax;
4     uint32_t ebx;
5     uint32_t ecx;
6     ......
7  } cpu_regs_t;
8
9  typedef struct system_call
10 {
11    int syscall_num;
12    int syscall_args[ARG_NUM];
13    int pid;
14    char process_name[MAX_LEN];
15    ......
16 } system_call_t;
17
18 cpu_regs_t regs_info = intercept_syscall();
19 system_call_t syscall_info = syscall_handler(regs_info);
20 notify_management_vm(syscall_info);
```

Figure 2. System call interception and semantic reconstruction

Figure 2), such as system call number (`syscall_num`) and process identification (`pid`). Ultimately, system call information (`syscall_info`) is sent to the MVM by function `notify_managment_vm` (line 20 in Figure 2). In order to understand the events happened in the TVM, a monitoring tool in the MVM should perform semantic reconstruction, which is tight with guest OS in the TVM. That is, the tool should be OS-ware. If there is another TVM with different type or version, the same semantic reconstruction may not work.

B. Solution Overview

We adopt a novel method to encapsulate the OS-related information of a TVM with a monitoring driver. The monitoring driver is dynamically loaded when the corresponding TVM is launched on or migrated to this platform. The driver provides uniform interface for other monitoring and management tools, which extends the working scope of existing monitoring tools and provides efficiency for new tools.

The concept of monitoring driver is similar to the device driver in traditional OS. Before presenting the details of our solution, we briefly review the device driver model in traditional OS. Device driver is the integral component of OS, and provides the communication channel between computer system and peripheral equipments [20]. There are distinct difference of device driver model between Windows and Unix/Linux system. In this paper, we only introduce the Linux device driver model, as we take it as the reference model for VMDriver. Linux considers each device in the system as a file, and provides a simplified interface for user space applications. There are three types of device in Linux system, `character`, `block` and `network` [21].

`Character` and `block` device driver are accessed in the same way, but different with `network` device driver. `Character` and `block` devices are viewed as special files, and provide the same interface as other common files. Application can perform generic file operation on these devices through device driver, such as `open`, `read`, and `write`. Device driver masks the variation of diverse devices and bridges the gap between applications and peripheral equipments with uniformed interface. An application can access any type of existing or future device once its device driver is available in the OS.

For sake of flexibility, device driver in Linux is usually compiled as kernel module. When a device driver is registered into the kernel, the dispatch routine about the specific device is represented through structure `file_operations`, and exports all public functions through a symbol table. These routines implement the actions that all applications can perform on this device. Device driver maps file operations to the handler of the device.

We adopt the idea from Linux device driver model for VMDriver, as Figure 3 shows. Different shapes (e.g., ellipse, rectangle, and diamond) represent different types of monitoring driver in the MVM, which correspond to different guest OSes in TVMs. These drivers provide a standard interface for upper monitoring tools or other management applications. Event sensor in the VMM intercepts certain events which can be configured by each driver. Whenever an event is intercepted by event sensor, the low level information is intercepted and its semantic information is reconstructed by the driver. Thus, the tools and applications in the MVM can be independent of guest OS.

Through this way, VMDriver separates the monitoring point from the semantic reconstruction module and masks the variance of guest OS for general but fine-grained monitoring mechanism. A monitoring driver is implemented as a kernel module in the MVM, which is similar to the device driver in traditional Linux. The responsibility of the management module in the MVM is to control all monitoring drivers. When needed by a monitoring tool, a monitoring driver is loaded into the MVM's kernel by the management module.

This paper makes an analogy between the device driver model and VMDriver from three aspects: object, target, and procedure. There are many TVMs with various categories and versions, which are similar to diverse devices in OS. The device driver masks the heterogeneity of underlying devices. VMDriver uses monitoring driver to shield the differences of TVMs. The procedure is similar: both provide transparency to upper software (applications vs. monitoring tools), and can be dynamically loaded when needed.

C. Design Requirements

Aiming to provide practical and general monitoring mechanism for large-scale system, VMDriver should be:

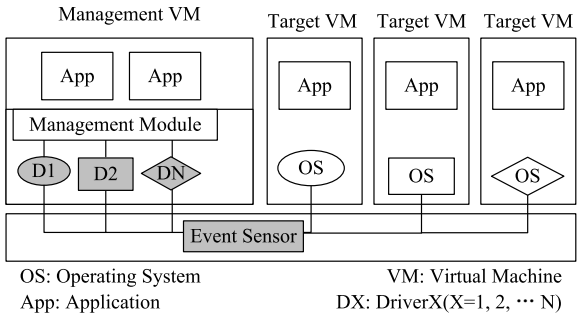


Figure 3. VMDriver architecture for VM monitoring

comprehensive–it inspects all TVMs on a physical platform; independent–it should be independent from the type and version of TVMs, and uniform–it provides common interface for other services and management tools. In order to achieve these requirements, we identify the following challenges in our design of VMDriver.

Separation and cooperation: All existing security monitoring methods are deployed monitoring function in VMM layer, which leads to the inspection for specific guest OS. For the purpose of easy implementation and high performance, event sensor and semantic reconstruction are all implemented in VMM level in previous work. However, they lack flexibility for general virtualized platform. We believe that for the sake of generality, it is critical to shift semantic reconstruction from VMM to MVM, while event interception is still deployed in VMM for effectiveness and performance purpose. Therefore, building efficient cooperation between event inspection and semantic reconstruction is critical for the practice of VMDriver. When an event happens in a TVM, event sensor acquires the state with low level data (for example registers or memory pages) and notifies the monitoring driver in the MVM. During this process, the monitoring driver needs the help of VMM, e.g., to obtain the virtual memory page of the TVM.

Universal interface: To enable the generality of monitoring function, a monitoring driver should provide standard interface for other monitoring tools. For the sake of flexibility, a monitoring driver should be encapsulated in the form of kernel module. Because VMM merely provides virtual hardware interface for these TVMs, a monitoring driver can not detect the type and version of a TVM automatically. That is, a high level monitoring tool in the MVM should specify the type and version of guest OS in order to load the specific monitoring driver. Necessary function in the MVM should be provided for this, e.g., via VM launch and migration daemon.

IV. DESIGN AND IMPLEMENTATION

We have implemented VMDriver on Xen 3.1 with Linux 2.6.18. There are two different virtualization modes provided

by Xen: para-virtualization and full-virtualization. A guest OS needs to be modified under para-virtualization mode for high performance, which is suitable for open source OS. Under full-virtualization mode, a guest OS can run on Xen without any modification, which usually needs support from hardware. For generality purpose, VMDriver aims to support as many OSes as possible including non-open source ones (e.g., Windows). Thus we focus on the monitoring of full-virtualization TVMs in this paper. Note that VMDriver can be easily implemented on other hypervisors, such as VMware and Virtual PC.

A. Implementation Overview

In Xen, the MVM is a privileged VM called `Dom0`, which is responsible for managing and controlling other unprivileged VMs called `DomU`. Figure 3 shows the software stack of VMDriver on Xen. From bottom to top, the stack includes event sensor, diverse monitoring drivers (Drive1, ..., DriverN), management module, and monitoring tools. Event sensor in the VMM intercepts the variation of a `DomU`'s state and provides communication interface for monitoring drivers in `Dom0`. When specific event happens in `DomU`, event sensor intercepts it and notifies `Dom0` with this communication interface. On the other side, individual monitoring driver corresponding to the type and version of `DomU` invokes the communication interface to map virtual address from `DomU` during semantic reconstruction. The management module, which is independent from `DomU`, dynamically loads monitoring driver for high-level monitoring tool.

In general, there are large numbers of events happening in a TVM (`DomU`). For the interest of VM monitoring, we focus on system call, which are also the target of most existing security monitoring tools [22]. System call is the useful way to inspect process state, through which the system state of a TVM can be derived. Whenever a system call happens, the event sensor in VMM intercepts and turns the execution flow to the VMM. With the light of hardware-assisted virtualization technology such as Intel VT and AMD SVM, our event sensor is transparent to `DomU`. We use Intel VT in our implementation; however our technology can be easily ported to AMD platform.

B. Event Sensor in VMM

Both Intel VT and AMD SVM adapt privileged instructions with a new CPU execution mode that allows the VMM to run in a new `root` mode below ring 0. The control flow transferring into the VMM is called `VMExit` and the control flow transferring to the TVM is called `VMEntry`. A VM can explicitly force a `VMExit` by using a `VMCall` instruction. A guest OS runs in `non-root` mode, which ensures that critical OS operations cause `VMExit`, such as privilege instruction, external interrupt, and page fault. When this happens, the state of guest OS (registers and context) is stored in the virtual machine control structure

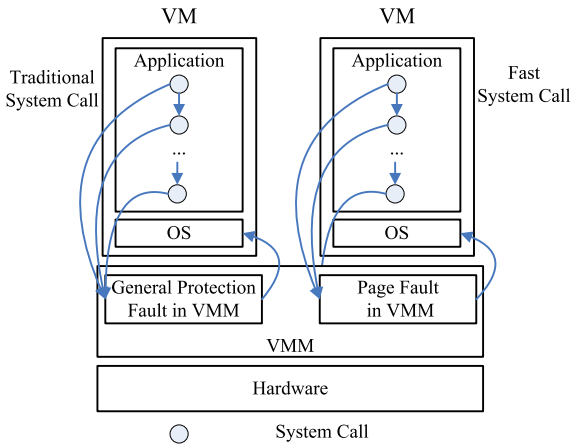


Figure 4. System call handler in VMM

(`VMCS`) region, which is maintained by the VMM. After that, the VMM checks the reason of `VMExit` and executes the corresponding handler. After the VMM handles the event, the CPU resumes the execution flow of guest OS. At the same time, the processor transforms from `root` mode to `non-root` mode.

In legacy Linux, system call is the interface between user mode (ring 3) and kernel mode (ring 0). There are two methods of system call implementation: software interrupt (`int $0x80`) and fast system call (`sysenter` in Intel and `syscall` in AMD). With traditional software interrupt, when one process in user mode invokes a system call, the kernel checks its privilege and input parameter, and invokes the corresponding kernel mode routine. The transition from user mode to kernel mode is time-consuming. Fast system call provides a quick transition method. Because there is no privilege inspection and stack operation, its execution speed is faster than software interrupt. So far fast system call has been applied in Linux 2.6 and Windows XP. Related registers (`SYSENTER_CS_MSR`, `SYSENTER_EIP_MSR`, and `SYSENTER_ESP_MSR`) must be prepared before instruction `sysenter`/`syscall` executes. Register `SYSENTER_EIP_MSR` indicates the entry point of function `sysenter_entry`.

According to the implementation of different system call, we use different mechanisms to intercept them in the VMM. As Figure 4 shows, system call with software interrupt can be intercepted by hardware and handled by the VMM as general protection fault. With hardware supported virtualization, interrupt instruction is trapped by CPU and handled by the VMM. We replace the interrupt handler with event sensor in the VMM. When a system call is invoked by `int $0x80`, it is intercepted by event sensor before the interrupt handler.

In order to intercept fast system call, we employ the

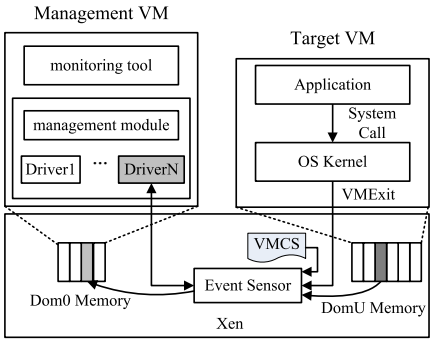


Figure 5. Communication between monitoring driver and event sensor

page fault handler mechanism of Xen. When the monitoring procedure starts, the entry address (SYSENTER_EIP_MSR) of fast system call is set to be non-existent value. When the application invokes a system call, page fault happens and is trapped by Xen, where the event sensor implements the interception. Intuitively, this mechanism increases the frequency of VMExit thus introduces overhead. However, VMExit is a very frequent operation in full-virtualization mode (round 100,000 times per second). Our experiments show that the performance penalty is tolerable (cf. Section V).

Event sensor in Xen is dependent on the underlying hardware and in effect for all guest OSes. It intercepts low level information (registers) about DomU, which is similar to the structure cpu_regs_t in Figure 2. Event sensor saves all information in local buffer and notifies the corresponding monitoring driver in Dom0, which performs semantic reconstruction.

C. Monitoring Driver

When a monitoring driver is loaded, it creates an event channel port and a shared page. The event channel port marks the communication way by which Xen notifies the monitoring driver. The shared page contains temporal information between Xen and the monitoring driver. For semantic reconstruction, the monitoring driver launches hypercall (do_domctl) to notify Xen. When the event sensor intercepts system information about DomU, it replicates the information (registers in VMCS and DomU kernel information) to Xen through function __hvm_copy_foreign. And then it copies these information to the shared page through function copy_to_user and notifies the monitoring driver through event channel port. Figure 5 illustrates the communication between the monitoring driver and event sensor. This notification and invocation procedure executes recursively until the procedure of semantic reconstruction is completed.

The major function of a monitoring driver is to translate binary data to OS object (e.g., process and file). Each monitoring driver corresponds to one type of guest OS. In this paper, we mainly introduce the semantic reconstruction procedure of three common OSes: Linux 2.6.24, Linux 2.6.31, and Windows XP.

For Linux, when a system call happens in DomU, the event sensor intercepts it and the control flow is transferred to Xen. The intercepted information includes register EAX which stores system call number and arguments hold in other registers (EBX, ECX, EDX, ESI, EDI). Kernel stack (SS and ESP) and structure thread_info share 2 pages in Linux. Structure thread_info locates in lower address, while the kernel stack locates in higher address. That is, if the page size is 4KB, structure thread_info aligns to 8KB (2^{13}) and the lower 13 bit is set to zero. The current position of kernel stack is denoted by register ESP, which can be acquired from the VMCS field. Based on the discussion above, the initial address of structure thread_info can be calculated by masking the lower 13 bits of register ESP, ESP&($\sim (2^{13}-1)$). By this means, the pointer of structure thread_info is reconstructed.

Structure thread_info and task_struct contains almost complete process information in DomU in the form of doubly-linked list, which is described in detail by [20]. Through this way, all processes in DomU can be displayed. In addition, we analyze all file objects that the current process operates on including all opened network connections. We focus on four types of system information of DomU: (1) single process information, (2) process list in DomU, (3) file objects that the current process operates on, and (4) all network connections. There is a little difference for the kernel structure between Linux 2.6.24 and Linux 2.6.31, such as process structure (task_struct). The offsets of these items in the structure differ with each other, and we define these structures in distinct kernel modules.

For Windows XP, the semantic reconstruction process has some differences. Figure 6 illustrates the information about process in Windows. Usually, there are four structures about Windows process: EPROCESS, Process Environment Block (PEB), ETHREAD, and Thread Environment Block (TEB). Virtual address 0xFFDFF000 contains the pointer of structure KPCR. Structure KPCR has the member of KPRCB. Structure KPRCB is a composite structure which contains the current thread structure (KTHREAD). Structure KTHREAD is the first member of structure ETHREAD. Structure ETHREAD has the member ThreadsProcess, which represents the process that the current thread belongs to. Structure EPROCESS has one member which points to the next process structure. Through these, we can traverse all processes running in one TVM. Although the implementation detail is different, the fundamental principle of semantic reconstruction is similar to that in Linux.

D. Management Module

The unified interface is provided by a global management module called manage_mod, which is implemented as a

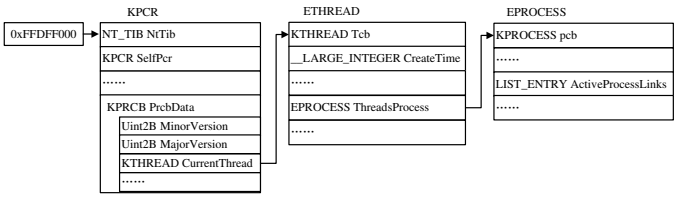


Figure 6. Process structure in Windows XP

kernel module and loaded when `Dom0` starts. It then loads a monitoring driver whenever a monitoring tool requires. In order to work with other modules, `manage_mod` complies with standard device driver interface. The main functions of module `manage_mod` are as follows.

1) `manage_mod_open`: A monitoring tool opens the management device `/dev/manage_mod` with this function, which returns the file descriptor (`fd`) of the management module.
2) `manage_mod_read`: After the semantic reconstruction of a monitoring driver in kernel mode is completed, the monitoring tools in user mode can obtain the result through this interface.
3) `manage_mod_ioctl`: This function loads a monitoring driver. According to the arguments from user space, it firstly checks if a `DomU` with `domid` exists. If yes, it loads the corresponding monitoring driver for the guest OS in the `DomU`. Because Xen works on the hardware merely, it is impossible to detect the type and version of guest OS in Xen or `Dom0`. In order to solve this problem, a monitoring tool should specify the type and version it intends to monitor. Through this interface, an administrator or management tool can specify the process and file to be monitored, which is very useful in real world. Our current implementation supports four types: (1) `LIST_PROC_INFO`: list the detailed information about specific process, such as `pid`, `uid`, `gid`, process name, process state, opened files, and network connections; (2) `LIST_PROC_LIST`: display all processes running in the `DomU`; (3) `LIST_FILE`: list all operations on specific files, such as `pid`, process name, system call number, file name, and operation time; (4) `LIST_NET_CON`: display all network connections in the `DomU`. We can develop other security functions in monitoring drivers.
4) `manage_mod_close`: This function closes the management device with file descriptor (`fd`) returned by function `manage_mod_open`.

V. EXPERIMENTS AND EVALUATION

We test both the functionality and performance of VMDriver. The functionality is proved by verifying if monitoring drivers can provide desired monitored information about these TVMs with various guest OSes. The performance impact is measured with CPU processing time and I/O speed with our experiments. We discuss the advantage and limitation of VMDriver at the end of this section.

```
// define the function request structure
typedef struct request
{
        int domid;
        int os_type;
        char *arg;
} request_t;

// open the management device
fd = open("/dev/manage_mod", O_RDWR);
if (fd == -1)
        return ERROR;

// initialize the request structure
request_t req;
req.domid = argv[1];
req.os_type = get_os_type(argv[2]);
req.arg = NULL;

// send ioctl request
ioctl ( fd, LIST_PROC_LIST, &req);

// get the response information and output
while( read(fd, buffer, BUFSIZE ) != 0 )
{
        printf("%s\n",buffer);
}

// close the management device
close(fd);
```

Figure 7. The sample monitoring application

A. Functionality

In order to demonstrate the generality of VMDriver, we have implemented three monitoring drivers for three different guest OSes: Ubuntu 8.04 with Linux 2.6.24 (`Dom1`), Ubuntu 9.10 with Linux 2.6.31 (`Dom2`), and Windows XP (`Dom3`). VMDriver runs in the kernel mode of `Dom0`, thus does not affect the normal execution flow of other `DomU`'s.

Figure 7 shows a test program (considered as a monitoring tool) which we use to display the process list in each TVM. Domain identification (`domid`) and OS type (`os_type`) are the arguments represented by the request structure `request_t`. This application opens the management device with `open("/dev/manage_mod", O_RDWR)`, which invokes function `manage_mod_open` of the management module in kernel mode. Structure `request_t` is defined and initialized for specific domain (`Dom1`). Function `ioctl` specifies the service type `LIST_PROC_LIST`, which lists the information of all processes in `Dom1`. After that, this application fetches all process information recursively. The same function is used to fetch process information of other TVMs with different

```
[root@localhost-120 monitorNew]# xm list
Name                        ID   Mem VCPUs      State   Time(s)
Domain-0                     0   915     8     r-----    389.8
Windows                      3  1024     1     r-----   2552.4
ubuntu-8.04                  1  1024     1     -b----    127.6
ubuntu-9.10                  2  1024     1     -b----     62.2
[root@localhost-120 monitorNew]# ./list_process 1 LINUX_2_6_24
process name          pid    uid     state
gconfd-2              5061   0       INTERRUPTIBLE
gnome-keyring-d       5063   0       INTERRUPTIBLE
x-session-manag       5064   0       INTERRUPTIBLE
seahorse-agent        5117   0       INTERRUPTIBLE
dbus-daemon           5125   0       INTERRUPTIBLE
gnome-settings-       5126   0       INTERRUPTIBLE
pulseaudio            5130   0       INTERRUPTIBLE
gconf-helper          5131   0       INTERRUPTIBLE
metacity              5143   0       INTERRUPTIBLE
gnome-panel           5146   0       INTERRUPTIBLE
```

(a) Process list in Dom1

```
[root@localhost-120 monitorNew]# ./list_process 2 LINUX_2_6_31
process name          pid    uid     state
gnome-settings-       1154   0       INTERRUPTIBLE
seahorse-daemon       1162   0       INTERRUPTIBLE
gvfsd                 1164   0       INTERRUPTIBLE
notify-osd            1168   0       INTERRUPTIBLE
gvfs-fuse-daemo       1170   0       INTERRUPTIBLE
metacity              1176   0       INTERRUPTIBLE
gnome-panel           1211   0       INTERRUPTIBLE
nautilus              1212   0       INTERRUPTIBLE
bonobo-activati       1214   0       INTERRUPTIBLE
python                1217   0       INTERRUPTIBLE
trashapplet           1222   0       INTERRUPTIBLE
```

(b) Process list in Dom2

```
[root@localhost-120 monitorNew]# ./list_pross 3 WINDOWS_XP
name        pid       priority  state      active_threads
IEXPLORE    740       8         Running    10
wuauclt     984       8         Waiting    4
cmd         1540      8         Waiting    1
conime      1080      8         Waiting    1
System      4         8         Running    51
smss        304       11        Waiting    3
csrss       404       13        Running    11
winlogon    428       13        Running    16
services    484       9         Running    15
lsass       496       9         Running    17
svchost     696       8         Running    19
```

(c) Process list in Dom3

Figure 8. Process list acquired by VMDriver

input arguments. Figure 8 shows the results. During the monitoring procedure, the management module loads the monitoring driver corresponding to guest OS.

We further test the generality of VMDriver by detecting rootkit in different DomU through the comparison between internal and external process list. We use adore-ng [23] as a sample, which is a Linux kernel rootkit that replaces certain kernel-level function pointers to hide some processes or files. In our test, adore-ng.ko is firstly loaded into Dom1 (Ubuntu 8.04). A user-level application named ava is used to control the functionality of adore-ng, i.e., to simulate an attacker who wants to conceal the existence of the victim's shell with the command ava i 5168 (5168 is the pid of bash in Figure 9). After that, we can find that there is no bash process in Dom1. However, the monitoring tool in Dom0 can discover the hidden process bash in Dom1, and Figure 9 shows the detection result. With the same experiment, the monitoring tool can detect adore-ng in Dom2, although guest OS is different.

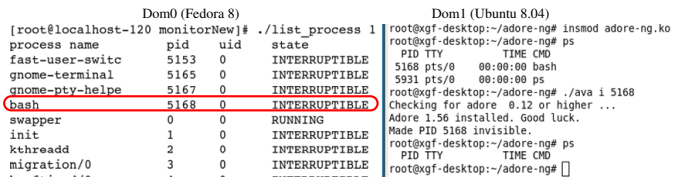


Figure 9. Identifying hidden process in Dom1 with monitoring tool in Dom0

FU [24] is a Windows kernel-level rootkit with similar function as adore-ng. Through the same method, our test tool can detect it with the unified interface provided by VMDriver. The only difference is the argument used by the test tool, which indicates a different monitoring driver is used to monitor the TVM running Windows XP.

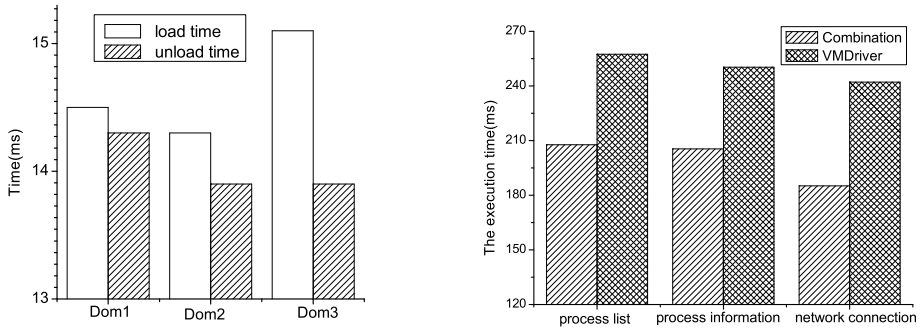
B. Performance

We study the performance of VMDriver with CPU-intensive and IO-intensive benchmarks. The test machine has two Intel Xeon E5310 CPUs running 8 cores at 1.6GHz, 4MB cache per core, and 4GB system memory. The host OS runs Fedora Core 8 (Dom0) and Xen 3.1.0. We use the same DomU as aforementioned. In order to examine the performance overhead, each DomU is allocated with one virtual CPU (VCPU) and 1024MB memory. For each test case, we run 10 times and take their average value.

Firstly, we measure the load/unload time of monitoring driver module. When a monitoring tool in Dom0 starts, it invokes the management module to load the corresponding monitoring driver. Figure 10(a) shows the measurements. The load/unload time of monitoring driver for these TVMs (Dom1, Dom2, Dom3) are all at millisecond level. Since the monitoring module is loaded once, the overhead of this operation is negligible during the monitoring procedure.

We then compare the execution time between traditional method (Combination) (which puts event sensor and semantic reconstruction together in the VMM) and VMDriver with three basic functions: (1) traversing process list, (2) printing single process information, and (3) displaying all network connections. Figure 10(b) shows the execution time of these functions. The results show that the performance penalty of VMDriver is from 21.8% to 30.7%. The reason is that during semantic reconstruction of VMDriver, DomU is suspended by Xen. There are multiple mode switches between monitoring driver in Dom0 and Xen, which do not exist in traditional method. The procedure of semantic reconstruction is the performance bottleneck, and it will be optimized in future work.

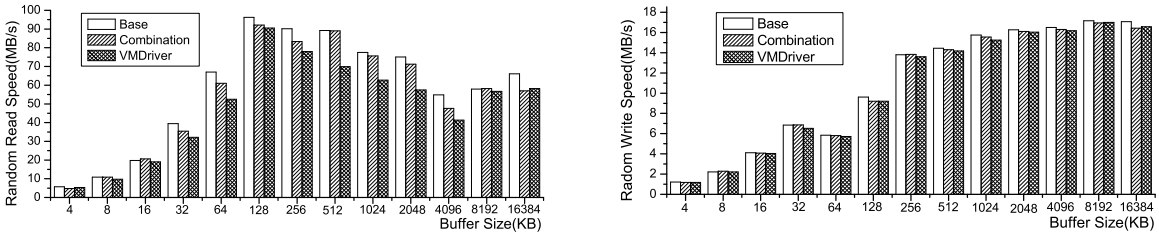
In order to test performance impact on input/output, we measure file read/write operations under three conditions: unmodified Xen (Base), combination of event sensor and semantic reconstruction (Combination), and separation of event sensor and semantic reconstruction (VMDriver).



(a) Load/unload time of monitoring driver module

(b) Execution time of three basic functions

Figure 10. Benchmark results of monitoring driver module



(a) `random-read` speed by `iozone`

(b) `random-write` speed by `iozone`

Figure 11. Benchmark results of I/O

We use `iozone`, a widely used benchmark for file system. The command we use is `iozone -a -i 0 -i 1 -i 2 -s 1G -f iozone.txt`, which indicates that the benchmark operates on file `iozone.txt`. We test the efficiency of file `read`, `write`, `random-read`, and `random-write`. The file size we chose is 1GB, which is as large as `DomU`'s memory. Figure 11(a) shows the speed of `random-read` when the block size varies from 4KB to 16384KB, and Figure 11(b) represents the speed of `random-write`. The horizontal axis represents the buffer size (KB), and the vertical axis represents the reading or writing speed (MB per second). When the block size increases from 4KB to 16384KB, the overall efficiency of `random-read` and `random-write` improves gradually. According to the result from Figure 11(a) and 11(b), we can find that the `random-read` speed is higher than `random-write`, and the monitoring function takes less effective on `random-write` than `random-read`. In the worse case, the efficiency loss is 26.1% for `random-read`, and 5.7% for `random-write`, compared with Base. In most cases, VMDriver has approximate efficiency compared with Combination. This proves that VMDriver brings little performance overhead.

C. Discussion

Lares [14] is an architecture for secure active monitoring, which disposes hooks in the TVMs protected by the VMM. For each guest OS, VMM should be modified, thus it limits the monitoring generality for diverse guest OSes. Ether [11] puts both event sensor and semantic reconstruction in the VMM, thus it does not solve the generality problem either. To the best of our knowledge, VMDriver is the first approach to address the generality problem of monitoring diverse guest OSes. At the same time, it enables fine-grained monitoring by separating event sensor and semantic reconstruction.

Our implementation assumes that Xen and `Dom0` are the trusted computing base (TCB) of a platform. However, as `Dom0` is a complete of Linux system, it may have the same vulnerabilities as legacy Linux OS. Thus, a new attack (VM escape) in `DomU` can utilize the vulnerability of Xen or management tools to penetrate into Xen or `Dom0`. In order to enhance the robustness of VMDriver, we implement most of modules in `Dom0`'s kernel mode, and `Dom0` does not provide extra service except management and monitoring tools.

We leverage underlying hardware with full-virtualization mode to intercept events in the VMM, as privileged opera-

tions must be completed by Xen which has higher privilege than all TVMs. This brings more performance overhead compared with native system or para-virtualization mode. While we minimize the performance overhead in our design, we rely on CPU vendors to further improve the performance of hardware-assisted virtualization.

VI. Conclusion

In this paper we propose VMDriver, a driver-based monitoring mechanism for virtualized platform. VMDriver separates event sensor and semantic construction in the VMM and the MVM respectively. Semantic construction is implemented by monitoring drivers and encapsulated as kernel modules, which correspond to variant TVMs. As a result, the monitoring mechanism is decoupled with the type and version of the TVM, and is effective for all types of guest OSes. We have implemented VMDriver on Xen with the light of hardware-assisted virtualization . Our experiments demonstrate the expected functionality and generality of VMDriver, and show that it brings acceptable performance overhead.

Acknowledgment

This work is supported by National 973 Basic Research Program of China under grant No.2007CB310900.

References

[1] S. Nanda and T. Chiueh. A Survey on Virtualization Technologies. Technical Report ECSL-TR-129, SUNY at Stony Brook, 2005

[2] J. E. Smith and R. Nair. The Architecture of Virtual Machines. IEEE Computer, vol. 38(5), pp. 32–38, 2005.

[3] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer, vol. 38(5), pp. 39–47, 2005.

[4] P. Barham, B. Dragovic, K. Fraser, S. H. T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. Xen and the Art of Virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 164–177, 2003.

[5] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne and J. N. Matthews. Xen and the Art of Repeated Research. In Proceedings of the 2004 USENIX Annual Technical Conference, pp. 135–144, 2004.

[6] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima and A. Mallick. Xen 3.0 and the Art of Virtualization. In Proceedings of the 2005 Linux Symposium, pp. 65–77, 2005.

[7] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection based Architecture for Intrusion Detection. In Proceedings of the 10th Network and Distributed System Symposium, pp. 191–206, 2003.

[8] X. Jiang and X. Wang. "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection, pp. 198–218, 2007.

[9] S. T. Jones, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In Proceedings of the 4th ACM/USENIX International Conference on Virtual Execution Environments, pp. 91–100, 2008.

[10] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 128–138, 2007.

[11] A. Dinaburg, P. Royal, M. Sharif and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 51–62, 2008.

[12] A. Lanzi, M. Sharif and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In Proceedings of the 16th Annual Network and Distributed System Security Symposium, 2009.

[13] C. Xuan, J. Copeland, and R. Beyah. Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection, pp. 304–325, 2009.

[14] B. D. Payne, M. Carbone, M. Sharif and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In Proceedings of the 2008 IEEE Symposium on Security and Privacy, pp. 233–247, 2008.

[15] VMware Home Page: http://www.vmware.com/.

[16] Virtual PC Home Page: http://www.microsoft.-com/windows/virtual-pc/.

[17] VMsafe Home Page: http://www.vmware.com/technical-resourcces/security/vmsafe.html.

[18] B. D. Payne, M. D. P. A. Carbone and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In Proceedings of the 23rd Annual Computer Security Applications Conference, pp. 385–397, 2007.

[19] A. Srivastava, K. Singh and J. Giffin. Secure Observation of Kernel Behavior. Georgia Institute of Technology Report, 2008.

[20] D. P. Bovet and M. Cesati. Understanding the Linux Kernel, 3rd Edition. OReilly, 2005.

[21] J. Corbet, G. Kroah-Hartman and A. Rubini. Linux Device Driver, 3rd Edition. OReilly, 2005.

[22] S. A. Hofmeyr, S. Forrest and A. Somayaji. Intrusion Detection using Sequences of System Calls. Journal of Computer Security, vol. 6(3), pp.151–180,1998.

[23] adore-ng: http://lwn.net/Articles/75991/.

[24] FU: http://rapiddigger.com/search/download-fu-rootkit/.