Lightweight Intrusion Detection of Rootkit with VMI-Based Driver Separation Mechanism

Chaoyuan Cui[1], Yun Wu[2], Yonggang Li[1], and Bingyu Sun[1]
[1] Institute of Intelligent Machines, Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, Anhui 230031 - China
[e-mail: cycui@iim.ac.cn,lygzr@mail.ustc.edu.cn, bysun@iim.ac.cn]
[2] Institute of Applied Technology, Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, Anhui 230088 - China
[e-mail: wuyun@rntek.cas.cn]
*Corresponding author: Yun Wu

Received September 6, 2016; revised December 12, 2016; accepted January 14, 2017; published March 31, 2017

Abstract

Intrusion detection techniques based on virtual machine introspection (VMI) provide high temper-resistance in comparison with traditional in-host anti-virus tools. However, the presence of semantic gap also leads to the performance and compatibility problems. In order to map raw bits of hardware to meaningful information of virtual machine, detailed knowledge of different guest OS is required. In this work, we present VDSM, a lightweight and general approach based on driver separation mechanism: divide semantic view reconstruction into online driver of view generation and offline driver of semantics extraction. We have developed a prototype of VDSM and used it to do intrusion detection on 13 operation systems. The evaluation results show VDSM is effective and practical with a small performance overhead.

Keywords: lightweight intrusion detection; introspection; semantic gap; driver separation mechanism; portability

A preliminary version of this paper appeared in IEEE ICC 2009, June 14-18, Dresden, Germany. This version includes a concrete analysis and supporting implementation results on MICAz sensor nodes. This research was supported by a research grant from the IT R&D program of MKE/IITA, the Korean government [2005-Y-001-04, Development of Next Generation Security Technology]. We express our thanks to Dr. Richard Berke who checked our manuscript.

1. Introduction

The development and promotion of cloud computing cause the virtual machine to become the new target of rootkit attack[1] [2] [3]. Rootkit is a piece of program code designed to elevate privileges, keep undetected and take control of a target machine without his consent. The common technique of rootkits is to hijack kernel control flow or manipulate kernel states in order to blind the system and prevent intrusion detection[4]. The latest generations of rootkits, such as programs that steal user names, passwords, and bank account information,are increasingly being used to make other malware more effective by hiding them from anti-malware tools. The teaming of malware with rootkits has been developed from intellectual challenge to commercial profit, and has caused rootkit developers to improve the quality and effectiveness of their stealth techniques dramatically [5]. Therefore, the intrusion detection and prevention of rootkit attack is becoming more and more challenging.

A threat report from McAfee shows that the rootkit is still continuing to grow rapidly. As highlighted in the report, there appeared about 300,000 new rootkit samples in 2014, which translates into nearly 820 new samples discovered every day[6]! And by the end of 2014, the total number of rootkits has reached 1.6 million samples. Moreover, antivirus companies are detecting literally thousands of new variants every day, and they are improving and constantly changing themselves in order to evade detection and removal [6] [7]. This alarming trend reveals the disturbing fact that existing rootkit intrusion detection system (IDS) fails to effectively cover the threat and keep up with the rootkit infections.

The general IDS of rootkit mainly adopted host-based architecture and the network-based architecture, which are commonly referred to as HIDS (Host Intrusion Detection System) and NIDS (Network Intrusion Detection Systems) respectively [8]. HIDS runs inside the end-system and is able to directly inspect the states and events of the target. Although achieving high visibility, HIDS is fundamentally limited in its isolation capability and thus has lower tamper resistance to prevent itself from being infected once the system is compromised. In contrast, NIDS is deployed outside of an end-system, achieving high attack resistance at the cost of poor visibility on the internal system states.

In order to analyze incidents, two broad families of methodologies, named signature-based detection and behavior-based detection, are available to combine with HIDS or NIDS architecture[9]. Signature-based detection identifies suspicious samples by comparing observed events with signatures of known attacks or predefined signatures. It is very accurately at detecting known threats. However, signature definition and matching may be expensive to perform. A related issue is that slight variants of well-known attacks cannot be effectively identified. Attackers can modify existing malware easily using countless methods to simply bypass publicly and/or privately available signature sets. Behavior-based detection builds a reference model of the usual behavior (e.g., processor usage level, band width consumption...) of the monitored system and looks for deviations from this model. It is effective at detecting previously unknown threats whenever attacks produce a deviation from the model of normal activity. However, behavioral-based systems are only as good as their policies.

In recent years, researchers change the rootkit defense landscape by leveraging recent advances of virtualization, and propose virtual machine introspection technology to build IDSes [10] [12] [13]. The vulnerable system runs as VM and VMI-based IDS pulls the monitoring tools from inside VM to outside. It directly accesses the hardware state and uses this information to analyze the inner state of VM. Because of isolation with VM, VMI architecture has high temper-resistance like HIDS. At the same time, VMI's direct observation of hardware offers high visibility comparable to that offered by an NIDS. So, VMI-based IDS provides a more robust view of target VM even in the face of OS compromise. Unfortunately, this is challenging due to the semantic gap, the difference between low-level bits of hardware and high-level semantic information of VM.

In this paper, we present a framework of intrusion detection with VMI-based driver separation mechanism (VDSM for short). The framework of driver separation is shown as **Fig. 1**. VDSM provides a light weight technique to bridge the semantic gap. The semantic view reconstruction is composed of offline phase and online phase. The former takes charge of semantic information extraction and builds static library of semantics, while the latter takes charge of semantic view generation and cross-view validation. The two operations are implemented by *BackDriver* and *FrontDriver* in VDSM, respectively. Both drivers are interrelated in function and independent on implementation.

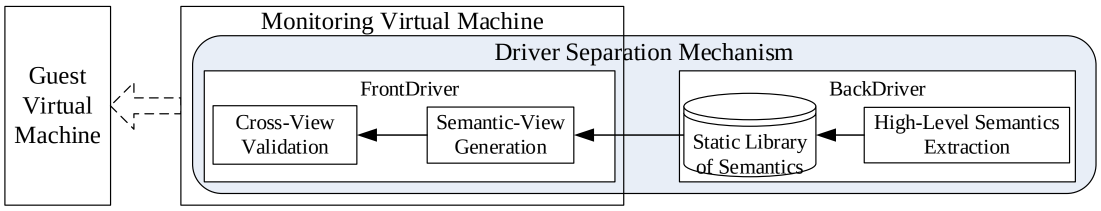

Guest Virtual Machine
Monitoring Virtual Machine
Driver Separation Mechanism
FrontDriver
Cross-View Validation
Semantic-View Generation
BackDriver
Static Library of Semantics
High-Level Semantics Extraction

Fig. 1. Framework of Driver Separation

VDSM can overcome the semantic gap challenge and improve the execution efficiency of intrusion detection. In summary, VDSM makes the following contributions:

- **Lightweight.** Based on the driver separation mechanism, the reconstruction of semantic view gets semantic information by directly calling functions encapsulated in ready-made static library, instead of parsing OS kernel frequently. The static library of semantics has been built offline before FrontDriver works. It can provide semantic information online instantly without waiting for semantic extraction. The driver separation mechanism sacrifices space for time simplifying online operations. The simplified handling can increase the speed of online processing and reduce the system load. Hence, our system is lightweight and the efficiency will be verified in later experiments.
- High-fidelity. Our VDSM monitors hardware state of VM, and interprets raw bits into semantic view according to hardware architecture and OS kernel data structure. As stated by Pfoh et al., this derived schema is the most reliable method [13]. The execution of any process requires hardware resources. As a result, it is impossible to bypass hardware. VDSM captures raw bits in hardware, instead of calling high-level application that may be cheated by malwares, for intrusion detection to ensure the authenticity of information. Furthermore, VDSM's strict isolation from the untrusted VM guarantees it's high temper-resistance. Any infected virtual machine can not affect the operation of other virtual machines because of isolation between them. In other words, VDSM can get real and detailed inner state of VM during view-generation.
- Portability. The static library containing semantic information is generated by BackDriver and the standard interface is also provided. Although BackDriver has to face all kinds of kernel version, it is executed in offline phase and does not interact with the end-user. The complete kernel semantics in the library can mask the implementation details between different kernel versions and provide a uniform type of semantic service. Hence, the static library can be widely used in virtualization platform such as Xen [23],

KVM [24], and QEMU [25]. The view generation through FrontDriver just interacts with static library. It is not sensitive to the kernel version and does not need to rely on any operating system knowledge.

The rest of this paper is organized as follows: Section 2 presents related work in VMI-based intrusion detection and semantic gap reparation. Next, Section 3 and 4 describe the design, architecture and implementation of our system. InSection 5, we evaluate the VDSM's effectiveness, performanceefficiency and portability. Finally, we summary our conclusions and outline our future research directions in Section 6.

2. Related Work

VMI provides strong isolation from the VM being monitored, and it has been widely used in intrusion detection [10][12] [14] [15]. However, the key challenge of VMI-IDS is how to eliminate the semantic gap. VMI-based intrusion detection is pioneered by Livewire [10], a framework to do security checking from outside the monitored VM. Livewire is capable of detecting certain kinds of attacks by inspecting VM's high-level semantic information, which is reconstructed by the tool of the Linux kernel crash dump(LKCD) [11].The hardware state obtained from LKCD is interpreted into OS events through the components of OS interface library. However, some inherent limitations exist because the OS interface library is tied to a particular guest OS and the achievement of LKCD is needed to compile the kernel with debugging symbols.

In the past several years, a number of researches have examined the inherent semantic gap challenge and implemented a number of introspection-based technique to mitigate it. Guest view casting technique of VMwatcher applies the knowledge of OS kernel, especially the semantic definition of kernel data structure, to interprets the binary resource observed from VMM [12]. However, such interpretation typically requires detailed, up-to-date knowledge of OS kernel. For example, to introspect the pid of a running process in a Linux kernel, one has to traverse the corresponding *task_struct* to fetch its pid field. The definition of *task_struct* is different with different versions of OS. So it is sensitive to OS kernel and not conducive to system transplantation.

In fact, acquiring such knowledge is often tedious and time-consuming even for an open source OS. Many other systems focus on exploring the application of VMI, and providing ready-made library function to ease the reconstruction of semantic view. LibVMI[16] and VIX [17] can be directly used to obtain processes or loaded modules of VM. LibVMI is developed to provide access to the interface of various information such as VM's internal process list, network port, opened file and loadable kernel module. However, the view-generation with libVMI or VIX has to refer the internal documents of the VM to achieve its function. For example, the process descriptor is located by the system.map file, which is the kernel symbol table of VM. This means that either libVMI or VIX is dependent to the VM knowledge and it's intelligence still needs to be improved.

AntFarm and Lycosid proposed a statistical technique to bridge the semantic gap [18] [19]. AntFarm counts the actual number of processes by tracking the process lifecycle (such as process creation, termination and switching) in a VM. It makes use of the paging mechanism and the memory management unit (MMU) in x86 and SPARC hardware architectures to identify running processes. The page global directory (PGD)base address of process address space stores in the CR3 register. The creation, termination and switching of process will change the value of the CR3 register. AntFarm determines the actual number of processes in

VM according to the number of PGD base address appearing in candidate list of CR3 register. As a follow-up study of AntFarm, Lycosid detects the existence of hidden processes in VM by means of hypothesis testing, then identifies them through CPU usage which is calculated through the least squares regression analysis. Unfortunately, process detection based on statistical inference may result in false negatives or false positives.

The above techniques take a great deal of understanding of hardware architectures and OS. As a result, the use of such approach will always be very specific. Recently, cutting edge research has begun to focus on stand-alone view-generation. Virtuoso [20] and VMST [21] are the typical methods that narrow the semantic gap through program code. Virtuoso is a VMI technique that is not sensitive to OS data structures. The basic idea of Virtuoso is to capture the instructions of system calls or applications in VM, and then generates introspection-aware security tools based on these instructions. The tools run out of VM and can achieve the same function as in VM. The code extraction and code mergence techniques on which Virtuoso relies do not guarantee 100% reliable, and can only be accessed the data which is obtained through the OS API, so the limitations of Virtuoso are obvious.

VMST utilizes online kernel data redirection technology to bypass the hardware status acquisition and analysis, and automatically enables an in-guest inspection program to become a VMI tool. VMST depends on system call, interrupt handing and context switch, which cannot be separated from OS knowledge, so it is sensitive to the kernel version. Moreover, VMST doesn’t support asynchronous system call and can’t read the memory data which swapped out to hard disk. In addition, VMST also does not support multicore guest virtual machine. It is not suitable to the wide range of VMI applications.

The produced semantic view is commonly used to analyze potential threat. Based on the semantic view, Livewire decide whether or not the system is compromised by means of its policy engine. VMwatcher scans the semantic view to examine possible security vulnerability using existing antivirus software outside VM. However, generality and efficiency of view-generation cannot be guaranteed. In contrast, VDSM overcomes these two weaknesses. Reconstruction of semantic view based on prepared library improves the portability as well as the online processing time.

3. System design

In this section we describe the design of our system. We will first present the major components of VDSM. Then we introduce the design issues of driver separation mechanism (DSM for short). In the next section we will delve into the particulars of VDSM, a prototype system that implements this framework.

3.1 Assumption and threat model

In this work, we assume an underlying x86 architecture running a hypervisor with two kinds of virtual machines: monitored guest virtual machine (GVM) and security virtual machine (SVM) in which our tools will be deployed. We also assume a trust worthy hypervisor, based on the observation that the source code of the hypervisor is much smaller and more reliable than the code in the existing OSes. Meanwhile, we assume a trusted hypervisor that provides VM isolation. This assumption is shared by many other VMI-based security research efforts [12] [14] [15].

In our work, the rootkit has root privilege access to compromise arbitrary entity and facility inside the GVM, including OS itself and applications. It can either modify any code or data, or aims at stealthily maintaining and hiding its presence in GVM to execute its malicious code.

However, based on the isolation of hypervisor, an attacker can only compromise OS kernel or application in GVM, it cannot break out of SVM and corrupt the underlying hypervisor.

3.2 VDSM overview

Fig. 2 shows the overview of our system. From Fig. 2, it can be seen that the environment of VDSM typically includes Virtual Hardware, Hypervisor, Virtual Machine and DSM component. In the following we describe them each.

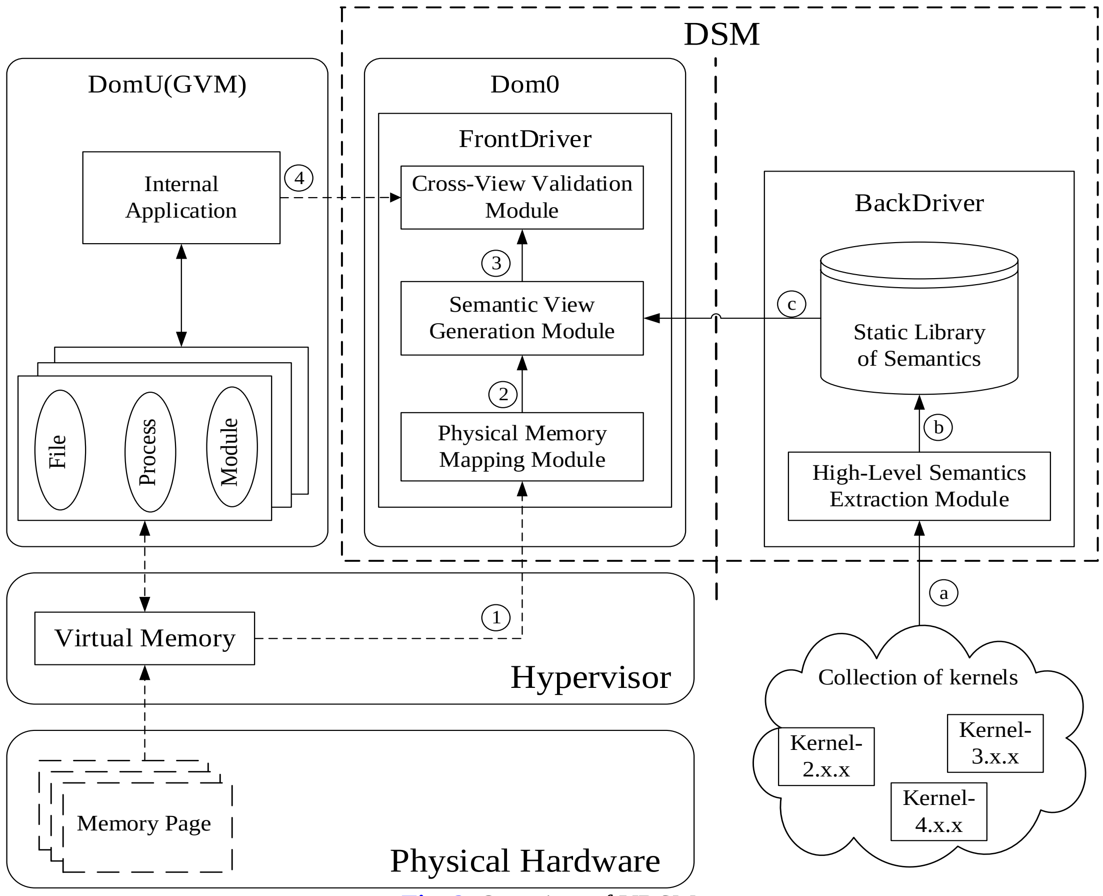


Fig. 2. Overview of VDSM.

Virtual Hardware: X86 is the most frequently virtual architecture used in today, and our work is based on the Intel x86 family of processors. The virtualization instructions set of Intel VT-x makes the x86 more effective, and provides the environment for running virtual machines. With the light of the technology of Intel VT, the DSM is transparent to GVMs in our implementation.

Hypervisor: Hypervisor sits between the OS and the underlying hardware, and acts as a bridge between the host and the guests. Hypervisor implements a hardware interface in software. The interface includes the microprocessor architecture as well as peripherals like disk, network, and user interface devices. The purpose of a hypervisor is to provide a virtual environment to run virtual machines.

Virtual Machine: The VMs running on hypervisor include several GVMs and one SVM. In our work, the GVM is assumed to be the attack target, and SVM is deployed with our security tools. SVM monitors the status of GVM through VMI technology and reports the suspicious behaviors.

DSM Modules: DSM is the main component of VDSM, it includes two parts of *FrontDriver* and *BackDriver*, implemented by several modules respectively. The details of DSM will be described below.

3.3 The Working Procedure of DSM

As illustrated in the **Fig. 2**, DSM is consist of two parts: the FrontDriver and the BackDriver. Their working procedure is described as follows.

FrontDriver interacts directly with end-user. It is used for reconstruction of semantic view and intrusion detection. After the task of intrusion detection starts, the Physical Memory Location Module firstly locates the hardware position of the current processes and the loaded modules in the GVM. Then the Semantic View Generation Module starts to build the high-level semantic view of GVM by parsing the physical memory through the Static Library generated by the *BackDriver*. Finally, based on the semantic views built in dom0 and domU, the Cross-View Validation Module examines the potential threats in GVM.

BackDriver must be completed before the start of *FrontDriver*. It plays the role of generating a Static Library of Semantics, which is used for parsing physical memory by *FrontDriver*. The execution of *BackDriver* is an offline operation, and implemented by means of High-level Semantics Extraction Module. It extracts the semantic information of OS internals (including running process list, system call and loaded modules) of GVM by compiling kernel, then encapsulates all these information into a static library and provides standard call interfaces. The static library is used to receive request from FrontDriver and returns the corresponding semantic information. Note that the BackDriver covers different versions of OS kernel.

4. System Implementation

We have implemented our prototype system of VDSM on x86 architecture and Xen 4.1.2 platform. For generality purpose, VDSM is able to support a variety of Linux distributions. In the following, we describe the implementation details, with a focus on GVM state procurement and semantic view reconstruction. Note that VDSM can be easily implemented on other hypervisors, such as KVM, Qemu and VirtualBox.

4.1 Layout of GVM

The raw memory allocated to GVM can be procured by way of hypervisor. However, the challenge is that it requires accurate layout of GVM kernel to understand what the physical memory pages are represented. The layout of GVM is mainly expressed by means of running processes, loaded modules, opened files, and so on. The process is the most important one to describe the state of GVM, and we mainly focus on process analysis in this work. Every process in OS is represented by a process control block [22]. Specifically, it is defined as *task_struct* in Linux, which is found in the <linux/sched.h>. All running processes are linked by a doubly linked list, each process has its own kernel stack which is allocated with the creation of process.

As shown in **Fig. 3**, kernel stack is a 8KB memory area in two consecutive page frames with the first page frame aligned to a multiple of 213. It contains a small data structure linked to the process descriptor, namely the *thread_info* structure, and the kernel code process stack. The *thread_info* structure and the *task_struct* structure are mutually linked by means of the fields task and stack, respectively. The figure also shows that the ESP register is a CPU stack pointer used to address the stack's top location. The kernel can easily obtain the address of the thread_info of the process currently running on a CPU from the value of the ESP register.

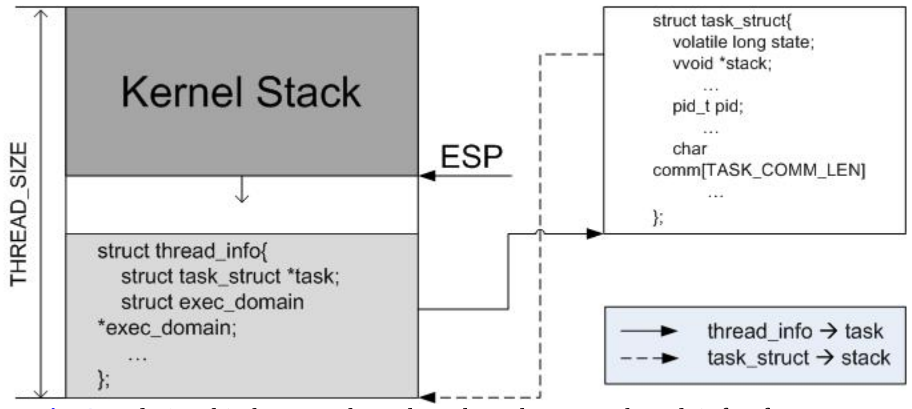


Fig. 3. Relationship between kernel stack, task_struct, thread_info of aprocess.

While the *task_struct* structure contains all the necessary information for representing a process, such as process ID (pid), process name (comm), process state (state), process memory management information (mm), list of open files, and pointers to the next and previous process in the list. Following this pointer, we can further parse the raw memory image and traverse the whole doubly linked list to reconstruct detailed semantic information of each running process. **Fig. 4** shows the list relationship between these *task_struct* structures. From the same memory image, we can also reconstruct a number of other important kernel data structures (e.g., the system call table, the interrupt descriptor table, and the kernel module list). In this research, we mainly focus on process tracking.

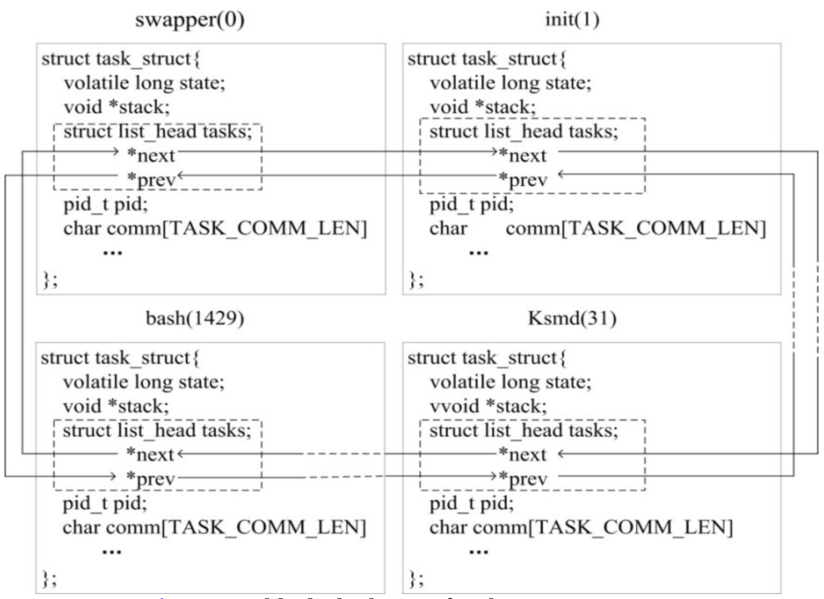


Fig. 4. Doubly-linked List of task_struct structure.

4.2 Xen memory management

VDSM parse the underlying physical raw memory allocated to GVM and reconstruct high-level semantic view. However, the main obstacle is to solve the semantic gap between raw memory and OS internals.

Xen is consist of virtualization layer and virtual domain[23]. The virtualization layer consists of hypervisor and privileged VM. In terms of Xen, each VM system is a domain. The privileged VM is called Dom0 and the others are called DomU. Dom0 is established with the start of Xen, it has privilege of direct access to the underlying hardware devices, and manages other domains through interface offered by hypervisor. DomU is generated through Dom0 and manages the physical resources through the hypervisor. In our system, *FrontDriver* of DSM is located in Dom0, and cannot directly get the details of isolated GVM of DomU. The reconstruction of semantic view requires address translation of binary resources observed from hypervisor.

Xen manages hardware resources through hypervisor. In order to provide VMs with zero-based and continuous memory space, hypervisor introduces a new layer of address space, named VM physical address space, between VM virtual address space and host address space. Therefore, two address translations are needed when access to physical memory: from guest virtual address (GVA) to guest physical address (GPA) translation and from GPA to host physical address (HPA)translation. That is to say the translation of GVA ->GPA->HPA is necessary. Fortunately, The virtual memory management unit(VMMU) of hypervisor uses a technology called Shadow Page Table (SPT) to achieve the direct translation of GVA ->HPA, hence improve the efficiency of memory access.

4.3 System implementation

As mentioned in Section III, VDSM is designed to perform lightweight detection of rootkit. As a result, our prototype is implemented in two phases: offline phase that generates static library of GVM OS kernel, online phase that reconstructs semantic view and perform intrusion detection. The static library is used to provide high-level semantic information of GVM for parsing raw memory during online phase. The two phases are corresponding to *BackDriver* and *FrontDriver* in **Fig. 2**, respectively .

4.3.1 Static library of OS kernel

The static library of OS kernel mainly contains two parts: the semantic information of kernel and some functions. The semantic information includes some data types and offsets of certain entries in specific data structures such as *task_struct* and *mm_struct*. The offsets are used to locate the virtual address of GVM, and the data types are used to determine how many bytes to be read in physical memory and which kind of high-level semantic information should be reconstructed outside GVM. The functions in static library are used to generate semantic information and capture specific physical memory.

Fig. 5 shows the generation of static library. In this algorithm, The GetOffset function (lines 5-8) returns the offset of the member in struct *task_struct*, such as pid, comm, tasks, etc. Note that this function should be written into a source file including specific header files that contains specific data structures such as *task_struct*. The source file is placed into kernel source for compiling and the returned results are data offsets we need. The ReadAddr function (lines 9-14) gets the address value of the first parameter src. The GetNext function (lines 15-19) is able to get the tasks address of the next process according to the tasks address of current process in the doubly-linked list. The Gettask_struct function (lines 20-26) is used to obtain the address of *task_struct*.

```
Require: Object task_struct defined in OS
Ensure: Static Library $\mathcal{L}$
typedef int IntType
typedef string MemberType
typedef unsigned int OffsetType
typedef unsigned long long AddrType
OffsetType GetOffset(MemberType& member)
{
    return &(((struct task_struct *)0) $\rightarrow$ member) ;
}
AddrType ReadAddr(AddrType* src, IntType width)
{
    AddrType dest=0;
    memcpy(&dest, src, width);
    return dest;
}
AddrType *GetNext(AddrType *addr)
{
    struct list_head *list = ((struct list_head *)addr) $\rightarrow$ next;
    return list;
}
AddrType *Gettask_struct(AddrType *addr_tasks)
{
    struct task_struct *task;
    struct list_head *list = ((struct list_head *)addr_tasks);
    task=container_of(list, struct task_struct, tasks);
    return task;
}
```

Fig. 5. Algorithm 1: generation of static library

Considering the principle of software reuse, we use a command to produce static library libsl.a, a common intermediate result. We can achieve an executable file by compiling libsl.a with the object programs in *FrontDriver*. The library makes the VDSM more modular, it is easier to recompile, and convenient to upgrade. The libsl.a contains all the functions that will be used in the *FrontDriver*. Using a function name and the appropriate parameters, the *FrontDriver* directly calls the function in libsl.a and obtains the corresponding information.

VDSM provides a generic, systematic methodology that can be applied to various OSes and virtualization platforms. Moreover, different versions of the same OS may have subtle variations for the same kernel-level data structure. The difference over the same OS adds additional complexity to semantic view reconstruction. However, the VDSM methodology remains effective despite these differences, as shown by our evaluation in Section 5.

4.3.2 Reconstruction of semantic view

In algorithm 2 (**Fig. 6**), the prepared static library and the user intention of end-user is necessary. By the way, user intention in this work refers to process trace. The reconstruction of semantic view works as follows. First, the physical memory corresponding to the current process is located by ESP of GVM stack pointer and SPT function (lines 2-4). Second, the static library is invoked to get the offset of tasks member in struct *task_struct*. Note that the hmatasks is the host machine address corresponding to the tasks, and the gva tasks is the guest virtual address of the tasks. Thus the address pointing to the next *task_struct* is determined (lines 5-8). Third, VDSM performs the traversal of process list (lines 9-19). For each process, VDSM gets the process ID, name and other information according to user intention specified by end-user, and the operation is done by ParseItem function (lines 11-16). Finally, information of all processes is achieved and output.

```
Require: Static Library $\mathcal{L}$, Intention Property $\mathcal{T}$
Ensure: Semantic View $\mathcal{SV}$
 1: $\mathcal{SV} := \emptyset$
 2: ESP := ctx $\rightarrow$ user_regs $\rightarrow$ esp
 3: gva_thread_info := $\sim$ (THREAD_SIZE - 1) & ESP
 4: hma_task_struct := SPT((AddrType)gva_thread_info, 0)
 5: gva_task_struct := $\mathcal{L}$.ReadAddr(hma_task_struct, int width)
 6: offset := $\mathcal{L}$.GetOffset(tasks)
 7: hma_tasks := SPT(gva_task_struct, offset)
 8: gva_tasks := $\mathcal{L}$.ReadAddr(hma_tasks, int width)
 9: while gva_tasks != END do
10:     gva_task_struct = $\mathcal{L}$.Gettask_struct(gva_tasks)
11:     for $item \in \mathcal{T}$ do
12:         offset := $\mathcal{L}$.GetOffset(item)
13:         hma_item := SPT((AddrType)gva_task_struct, offset)
14:         $v$ := ParseItem(hma_item)
15:         $\mathcal{SV} := \mathcal{SV} \cup v$
16:     end for
17:     hma_tasks := SPT((AddrType)gva_tasks,0)
18:     gva_tasks = GetNext(hma_tasks)
19: end while
20: return $\mathcal{SV}$
```

Fig. 6. Algorithm 2:reconstruction of semantic view

4.3.3 Cross-view validation

Cross-view validation for intrusion detection has been studied and implemented in various virtualization platform. The key aspect of cross-view validation approach is to compare the VMI-level semantic view with GVM OS-level view. In our system, right before we take the process snapshot, we run some commands, such as *ps*, *pstree* or *top*, and save their result to a file. Then we attach the snapshot and run VDSM in SVM to reconstruct semantic view based on the observation and analysis of GVM address space. Finally, the two output files are syntactically compared to identify the possible attack.

VDSM is deployed out of GVM and performed on virtualization level, so is trusted and has high temper-resistance. The extracted semantic view should not be compromised even if the rootkit is existed in GVM. In other words, VDSM improves the detection accuracy as well as temper-resistance without losing the visibility on internal system states.

5. Experiments and evaluation

In this section, we will evaluate VDSM using a number of different criteria: effectiveness, performance and portability. To evaluate the effectiveness of VDSM, we look at the high-fidelity of semantic view reconstruction and the accuracy of rootkit detection. Next we investigate the time performance overhead of BackDriver and FrontDriver, and further compare the efficiency of VDSM with several off-the-shelf anti-virus tools. Finally we will discuss the portability on the diversity of OS distribution and kernel version.

5.1 Experimental environment

We measured effectiveness, performance and portability on various of OS and kernel versions. The host machine contains Intel Core i7 processor with 4 cores running at frequency 2.93GHz, 4GB memory, and 1TB hard disk. The SVM OS is a 64-bit Ubuntu 12.04 with 3.2.0-23 linux kernel. The GVMs are different distribution of 64-bit linux OS and include a varietyof 13 kernel versions. All GVMs are conFig.d with 1 virtual CPU and 1GB memory. Table I lists the detailed configuration. In all experiments, the GVM are pinned to run on separate CPU cores in the host (using the Linux taskset command).

5.2 Effectiveness

Our goal is to provide an external intrusion detection technique, to achieve the same effect as that running in-VM. To this end, as presented in **Table 1**. We took 13 commonly used OS version to demonstrate its practicality and effectiveness of VDSM prototype. For convenience of explanation, we took Ubuntu12.04 as example to reconstruct external semantic view for initial process, new process and hidden process.

Table 1. Experimental environment

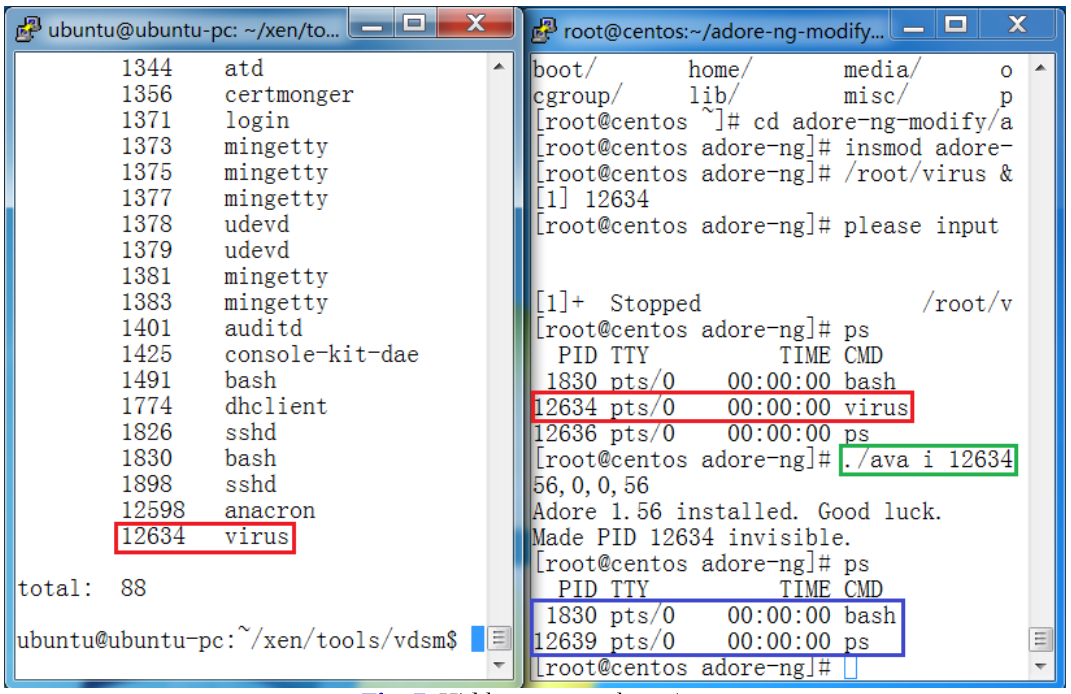
VM	CPU	Memory	OS	Dist Time	Kernel
SVM	4	4GB	Ubuntu12.04	2012.04.26	3.2.0-23
GVM11	1	1 GB	Ubuntu10.04	2010.04.29	2.6.32-21
GVM12	1	1 GB	Ubuntu12.04	2012.04.26	3.2.0-23
GVM13	1	1 GB	Ubuntu14.04	2014.04.17	3.13.0-24
GVM21	1	1 GB	Debian7.0.0	2013.05.06	3.2.0-4
GVM22	1	1 GB	Debian7.3.0	2013.12.16	3.2.0-4
GVM23	1	1 GB	Debian7.6.0	2014.07.14	3.2.0-4

GVM31	1	1 GB	CentOS6.3	2012.07.09	2.6.32-279
GVM32	1	1 GB	CentOS6.4	2013.03.09	2.6.32-358
GVM33	1	1 GB	CentOS6.5	2013.12.01	2.6.32-431
GVM41	1	1 GB	Fedora 16	2011.11.08	3.1.0-7
GVM42	1	1 GB	Fedora 17	2012.05.29	3.3.4-5
GVM43	1	1 GB	Fedora 18	2013.01.15	3.6.10-4
GVM44	1	1 GB	Fedora 19	2013.07.02	3.9.5-301

5.2.1 High-fidelity of semantic reconstruction

The first experiment observes the effect of semantic view reconstruction. VDSM can get not only the initial processes of GVM, but also the variation of processes. The initial process detection has been introduced in ModSG[26], an early stage of our research, so the presentation is omitted. Here we just describe the hidden process detection.

We describe our experiment with the adore-ng rootkit, an advanced Linux kernel rootkit that will directly replace certain kernel-level function pointers to hide files and processes [27]. **Fig. 7** is a screenshot showing an adore-ng infection. Within the **Fig. 7**, the right terminal window shows the inside of GVM, where the adore-ng kernel-level module (LKM) is first loaded (insmod adore-ng.ko). A user-level program called ava is used to control the LKMs functionality. Then, a backdoor daemon is executed (/root/demo/backdoor). After that, adore-ng is instructed to conceal the existences of a current process named *virus* whose pid is 12634 (ava i 12634). As revealed in the same terminal window, the outputs from the commands ps is already manipulated to conceal the existences of process with pid **12634.**


ubuntu@ubuntu-pc: ~/xen/to...
root@centos:~/adore-ng-modify...

```
        1344     atd
        1356     certmonger
        1371     login
        1373     mingetty
        1375     mingetty
        1377     mingetty
        1378     udevd
        1379     udevd
        1381     mingetty
        1383     mingetty
        1401     auditd
        1425     console-kit-dae
        1491     bash
        1774     dhclient
        1826     sshd
        1830     bash
        1898     sshd
        12598    anacron
        12634    virus

total:  88

ubuntu@ubuntu-pc:~/xen/tools/vdsm$
```

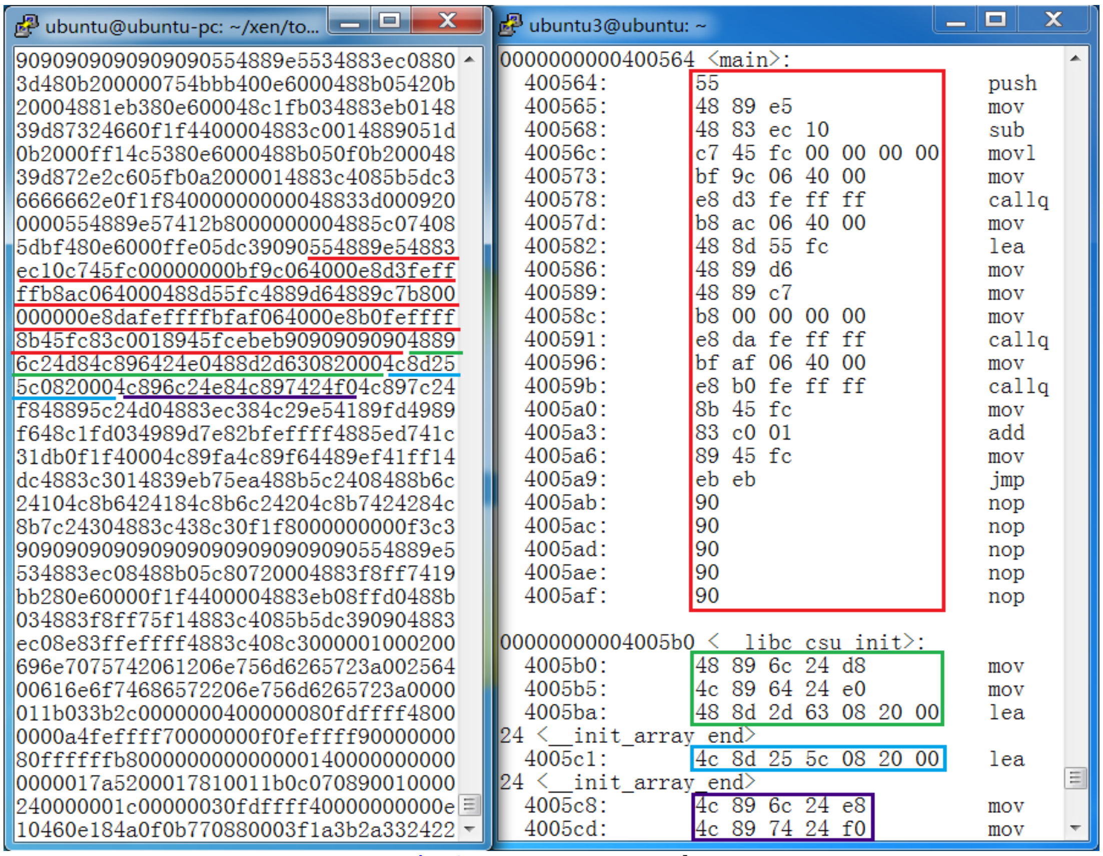
```
boot/          home/          media/        o
cgroup/        lib/           misc/         p
[root@centos ~]# cd adore-ng-modify/a
[root@centos adore-ng]# insmod adore-
[root@centos adore-ng]# /root/virus &
[1] 12634
[root@centos adore-ng]# please input


[1]+  Stopped                     /root/v
[root@centos adore-ng]# ps
  PID TTY          TIME CMD
 1830 pts/0    00:00:00 bash
12634 pts/0    00:00:00 virus
12636 pts/0    00:00:00 ps
[root@centos adore-ng]# ./ava i 12634
56,0,0,56
Adore 1.56 installed. Good luck.
Made PID 12634 invisible.
[root@centos adore-ng]# ps
  PID TTY          TIME CMD
 1830 pts/0    00:00:00 bash
12639 pts/0    00:00:00 ps
[root@centos adore-ng]#
```

Fig. 7. Hidden process detection

The external view of the GVM is shown on the left side of **Fig. 7**, and the window enumerates current running processes inside GVM. From the window, the internally-hidden process *virus* with pid 12634 is visible with VDSM.This experiment further demonstrates that the semantic view reconstructed by VDSM cannot be manipulated by the rootkit running inside GVM. As such, view comparison effectively exposes the existence of a rootkit (even if the hidden file and hidden process have unsuspected names).

Not only the hidden process can be found outside VM, the hidden file, module also can be found. Besides, all detail information about the hidden process can be captured including process code, mapped library, execution path and so on. Fig. 8 is the process code captured by VDSM. The left of the window is the code captured outside VM and the right is the code generated by disassembling (objdump -S xx) in GVM. Two kinds of code are completely same according to the left window. The captured code describing the action feature of the process can be used in memory forensics.



```
909090909090909090554889e5534883ec0880
3d480b200000754bbb400e6000488b05420b
20004881eb380e600048c1fb034883eb0148
39d87324660f1f4400004883c0014889051d
0b2000ff14c5380e6000488b050f0b200048
39d872e2c605fb0a2000014883c4085b5dc3
6666662e0f1f840000000000048833d000920
0000554889e57412b8000000004885c07408
5dbf480e6000ffe05dc39090554889e54883
ec10c745fc00000000bf9c064000e8d3feff
ffb8ac064000488d55fc4889d64889c7b800
000000e8dafeffffbfaf064000e8b0feffff
8b45fc83c0018945fcebeb90909090904889
6c24d84c896424e0488d2d630820004c8d25
5c0820004c896c24e84c897424f04c897c24
f848895c24d04883ec384c29e54189fd4989
f648c1fd034989d7e82bfeffff4885ed741c
31db0f1f40004c89fa4c89f64489ef41ff14
dc4883c3014839eb75ea488b5c2408488b6c
24104c8b6424184c8b6c24204c8b7424284c
8b7c24304883c438c30f1f800000000f3c3
909090909090909090909090909090554889e5
534883ec08488b05c80720004883f8ff7419
bb280e60000f1f4400004883eb08ffd0488b
034883f8ff75f14883c4085b5dc390904883
ec08e83ffeffff4883c408c3000001000200
696e707574206120e756d6265723a002564
00616e6f74686572206e756d6265723a0000
011b033b2c000000040000080fdffff4800
0000a4feffff70000000f0feffff90000000
80fffffb80000000000000014000000000
0000017a5200017810011b0c070890010000
240000001c00000030fdffff40000000000e
10460e184a0f0b770880003f1a3b2a332422
```

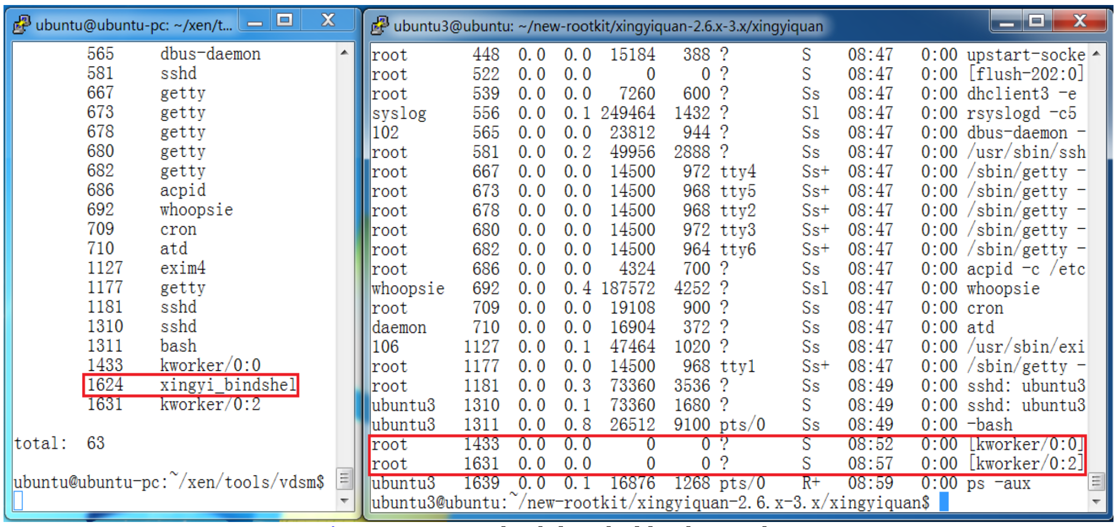
```
0000000000400564 <main>:
  400564:        55                      push
  400565:        48 89 e5                mov
  400568:        48 83 ec 10             sub
  40056c:        c7 45 fc 00 00 00 00    movl
  400573:        bf 9c 06 40 00          mov
  400578:        e8 d3 fe ff ff          callq
  40057d:        b8 ac 06 40 00          mov
  400582:        48 8d 55 fc             lea
  400586:        48 89 d6                mov
  400589:        48 89 c7                mov
  40058c:        b8 00 00 00 00          mov
  400591:        e8 da fe ff ff          callq
  400596:        bf af 06 40 00          mov
  40059b:        e8 b0 fe ff ff          callq
  4005a0:        8b 45 fc                mov
  4005a3:        83 c0 01                add
  4005a6:        89 45 fc                mov
  4005a9:        eb eb                   jmp
  4005ab:        90                      nop
  4005ac:        90                      nop
  4005ad:        90                      nop
  4005ae:        90                      nop
  4005af:        90                      nop

00000000004005b0 <__libc_csu_init>:
  4005b0:        48 89 6c 24 d8          mov
  4005b5:        4c 89 64 24 e0          mov
  4005ba:        48 8d 2d 63 08 20 00    lea
24 <__init_array_end>
  4005c1:        4c 8d 25 5c 08 20 00    lea
24 <__init_array_end>
  4005c8:        4c 89 6c 24 e8          mov
  4005cd:        4c 89 74 24 f0          mov
```

Fig. 8. Getting process code

5.2.2 Intrusion detection of rootkit

VDSM can bring up the capability of rootkit detection by external semantic view of GVMs. We have successfully identified 6 real-world rootkits by VDSM [28] [29]. On the country, some off-the-shelf antivirus software can hardly observe the existence of these rootkits. Compared with traditional antivirus software VDSM is deployed ouside GVM making it hard to be bypassed. Typically, rootkits hide kernel objects by hijacking the kernel function's control flow. For example, a rootkit named "xingyiquan" can hide a backdoor process (xingyi_bindshel) and specfic files (installation files and binary files) through hijacking operation funtions in "proc" file system. So the traditional scurity tools can not detect the hidden objects. Different with them, VDSM reads the information in hardware instead of calling functions in GVM. The detection results is shown as Fig. 9. The backdoor process xingyi_bindshel is hidden by the rootkit xingyiquan in GVM which is presented in right window. There is no process 1624 between 1433 and 1631. While, the hidden backdoor whose pid is 1624 can be detected by vdsm in SVM.


ubuntu@ubuntu-pc: ~/xen/t...
ubuntu3@ubuntu: ~/new-rootkit/xingyiquan-2.6.x-3.x/xingyiquan

```
        565     dbus-daemon
        581     sshd
        667     getty
        673     getty
        678     getty
        680     getty
        682     getty
        686     acpid
        692     whoopsie
        709     cron
        710     atd
        1127    exim4
        1177    getty
        1181    sshd
        1310    sshd
        1311    bash
        1433    kworker/0:0
        1624    xingyi_bindshel
        1631    kworker/0:2

total:  63

ubuntu@ubuntu-pc:~/xen/tools/vdsm$
```

```
root       448  0.0  0.0  15184   388 ?        S    08:47   0:00 upstart-socke
root       522  0.0  0.0      0     0 ?        S    08:47   0:00 [flush-202:0]
root       539  0.0  0.0   7260   600 ?        Ss   08:47   0:00 dhclient3 -e
syslog     556  0.0  0.1 249464  1432 ?        Sl   08:47   0:00 rsyslogd -c5
102        565  0.0  0.0  23812   944 ?        Ss   08:47   0:00 dbus-daemon -
root       581  0.0  0.2  49956  2888 ?        Ss   08:47   0:00 /usr/sbin/ssh
root       667  0.0  0.0  14500   972 tty4     Ss+  08:47   0:00 /sbin/getty -
root       673  0.0  0.0  14500   968 tty5     Ss+  08:47   0:00 /sbin/getty -
root       678  0.0  0.0  14500   968 tty2     Ss+  08:47   0:00 /sbin/getty -
root       680  0.0  0.0  14500   972 tty3     Ss+  08:47   0:00 /sbin/getty -
root       682  0.0  0.0  14500   964 tty6     Ss+  08:47   0:00 /sbin/getty -
root       686  0.0  0.0   4324   700 ?        Ss   08:47   0:00 acpid -c /etc
whoopsie   692  0.0  0.4 187572  4252 ?        Ssl  08:47   0:00 whoopsie
root       709  0.0  0.0  19108   900 ?        Ss   08:47   0:00 cron
daemon     710  0.0  0.0  16904   372 ?        Ss   08:47   0:00 atd
106       1127  0.0  0.1  47464  1020 ?        Ss   08:47   0:00 /usr/sbin/exi
root      1177  0.0  0.0  14500   968 tty1     Ss+  08:47   0:00 /sbin/getty -
root      1181  0.0  0.3  73360  3536 ?        Ss   08:49   0:00 sshd: ubuntu3
ubuntu3   1310  0.0  0.1  73360  1680 ?        S    08:49   0:00 sshd: ubuntu3
ubuntu3   1311  0.0  0.8  26512  9100 pts/0    Ss   08:49   0:00 -bash
root      1433  0.0  0.0      0     0 ?        S    08:52   0:00 [kworker/0:0]
root      1631  0.0  0.0      0     0 ?        S    08:57   0:00 [kworker/0:2]
ubuntu3   1639  0.0  0.1  16876  1268 pts/0    R+   08:59   0:00 ps -aux
ubuntu3@ubuntu:~/new-rootkit/xingyiquan-2.6.x-3.x/xingyiquan$
```

Fig. 9. Detecting backdoor hidden by rootkit

We have so far experimented with 6 linux rootkits, and view comparison-based scheme is able to detect all the rootkits tested and pinpoint the corresponding hidden processes. Due to the lack of space, we only present 6 of experiments in detail. Table II shows the detection results compared with the existing antivirus tools in GVM.

Table 2. Comparison with off-the-shelf antivirus tools

	adore-ng 0.56	kbeast v1	suterusu v1	f00lkit	diamorphine	xingyiquan
VDSM	√	√	√	√	√	√
Avast 1.3.0	-	-	-	-	-	-
AVG 2013.3118	-	-	√	-	√	-
Avira 7.6.0.10	-	-	-	-	-	-
chkrootkit 0.51	√	-	-	-	-	-
ClamAV 0.99.2	-	-	-	√	-	-
F-PROT 6.2.39	-	-	-	-	√	-
rkhunter 1.4.2	-	√	-	-	-	-

In experiment every rootkit was detected 100 times to test the detection accuracy. The results show that the detection accuracy is more than 98%. Because of the unconsistency of VM state[30], the detection result may appear garbled with a small probability. According to the description in [30] the unconsistency of VM state can be classified into intrinsic inconsistencies and extrinsic inconsistencies, and the former can not be avoided. Fortunately, VDSM is fast enough to reduce the probability of unconsistency of VM state.

5.3 Performance Overhead

In this section, we present the performance measurement results. We note that VDSM is operated outside of a VM. As a result, it will not affect the normal run of a GVM even when it is being examined. To evaluate the performance overhead of our system, we measure two different aspects: the execution time of VDSM as well as efficiency compare with some off-the-shelf antivirus software. In the following, we present two sets of measurement results.

5.3.1 Execution time of VDSM

The first aspect is to evaluate execution time of VDSM. To the end, we estimate the offline generation time of static library and the online scanning time of GVMs, respectively. In other words, we calculate the compilation time of *BackDriver* and the semantic view reconstruction time of *FrontDriver*. In particular, we choose all 13 GVM systems listed in **Table 1** to test the system efficiency of VDSM.

Fig. 10 shows the experimental result of offline performance. The horizontal axis represents 13 kinds of GVM OS environment, and the vertical axis represents the average value of 10 times compilation time corresponding to the GVM kernel. As shown in **Fig. 9**, we could see the time of generating static library is not more than 65 seconds. This shows the execution of *BackDriver* does not cause too much overhead. Moreover, it is an offline pre-processing operation, it would not affect performance of the online scanning of GVM.

Fig. 11 shows the reconstruction time of external semantic view. The horizontal axis represents the GVM systems, and the vertical axis represents the average value of reconstruction time. It is interesting to see that the scanning time of GVM is about 170 to 190 milliseconds. This proves that VDSM costs little performance overhead. On the other hand, the scanning time is so short that the guest user would not realize the pause and restart of GVM. So the online reconstruction of semantic view does not affect the user experience.

Moreover, we verify the performance overhead of scalability of VDSM. **Fig. 12** shows the execution time of external semantic view reconstruction with different number of processes in 13 kinds of OS environment. The horizontal axis is the number of processes, and the vertical axis is the average of 10 times semantic reconstruction time. From **Fig. 12**, we can conclude,(1) The semantic reconstruction time changes linearly with the number of processes. (2) Online semantic reconstruction does not affect the user experience. Even if the number of processes is 1000, online execution time is only about 300 milliseconds.

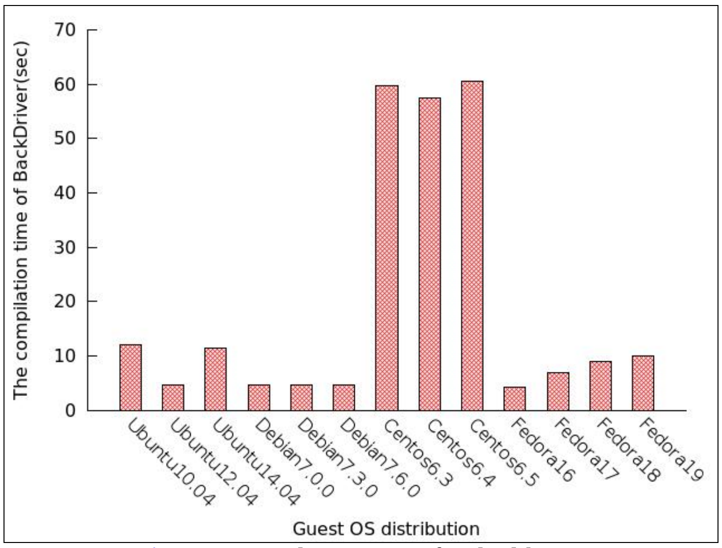


Fig. 10. Compilation time of stalic library.

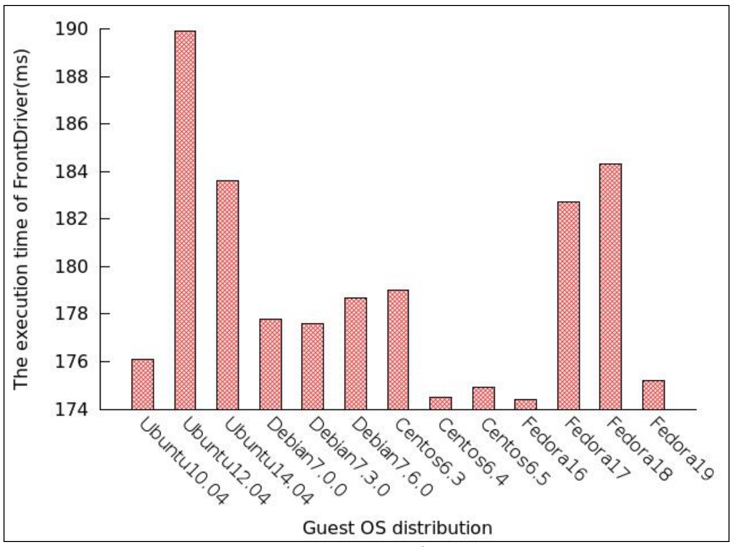


Fig. 11. The reconstruction time of external semantic view.

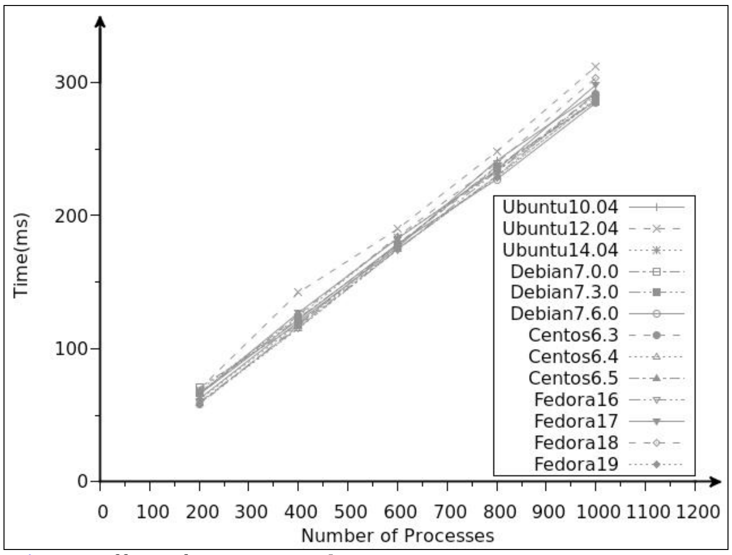


Fig. 12. Effect of process number on semantic view reconstruction.

5.3.2 Comparison with off-the-shelf antivirus tools

The second aspect of experiment is to compare the online scanning time of VDSM with several existed in-host antivirus software. In particular, we choose 7 different antivirus software systems and each one performs an internal scan. Our experiments are performed on Ubuntu12.04 GVM system, the GVM12 environment (1GB memory, 1 vCPU and 10GB disk), as shown in **Table 1**.

Table III reports the execution time of VDSM and some in-host antivirus tools. We can see that VDSM is the fastest and only takes 32 microseconds to perform intrusion detection. Here, the scanning time of VDSM include the time of semantic view reconstruction and cross-view validation. On the other hand, rkhunter is the most time-consuming in all of above tools and it takes about 24 minutes. Different from VDSM which just focuses on physical memory, rkhunter has more comprehensive scope of scanning. It scans the entire system and supports network port scanning beside rootkit signature. Although time-consuming, rkhunter is more effective than others. It is the only one to detect kbeast rootkit successfully, as is indicated in **Table 2**. The other antivirus software, such as avast, AVG, Avira, chkrookit, ClamAV and F-PROT, can either scan the entire system, or can scan a specified directory. In our experiments, these tools scan the installation directory of rootkits. The scanning time is less than 30 seconds.

Table 3. Comparison with in-host antivirus tools

	Scanning time
VDSM	32ms
Avast 1.3.0	401ms
AVG 2013.3118	12m27s
Avira 7.6.0.10	1s
chkrootkit 0.51	28.871s
ClamAV 0.99.2	8.617s
F-PROT 6.2.39	1s
rkhunter 1.4.2	24m15s

It is interesting to notice that the internal examination takes longer scanning time than VDSM, a result that sounds counterintuitive. However, considering the potential disk I/O slowdown introduced by virtualization as well as the fact that VDSM only examine the physical memory related to GVM processes, the shorter external scanning time of VDSM is actually reasonable.

Besides, we also comapred VDSM with oher out-of-box techniche. Table IV reports the comparison between VDSM and other out-of-box antivirus tools in three respects: performance, portability, and effctiveness. G means 'good'; A means 'average'; P means 'poor'. Because of the driver separation mechanism, VSDM need not interact with live kernel in semantic reconstruction. That is to say the complex action has been devided into two patrs and one part has been done before another starts. As a result VSDM has a higher performance. Moreover, the static library provides standard interfaces and contains all kinds of kernel version making VDSM more general.

Table 4. Comparison with out-of-box antivirus tools

	Performance	**Portability**	**Effectiveness**
VDSM	G	G	G
Livewire	P	P	A
VMwatcher	A	A	G
LibVMI	G	P	G
Antfarm	P	P	A
Virtuoso	A	G	P
VMST	A	P	G

5.4 Portability

Finally, we discuss how general (OS-agnostic) our design is, regarding different OS kernels. Our VDSM is implemented by the cooperation of *FrontDriver* and *BackDriver*. The key technique point of *FrontDriver* is to locate and parse the physical memory allocated to the target GVM, as described in algorithm 2.

Starting from kernel 2.6.20, Linux uses a global variable to store the current task. In particular, *thread_info* has a pointer to *task_struct*, while task_struct also has a pointer pointing to thread_info, which is usually allocated in the bottom of a kernel stack. Therefore, each time for the kernel to fetch the current task, it first gets the *thread_info* by bit-masking ESP with 8192 and then dereferencing the *task_struct* field in *thread_info*. In fact, almost systems adopt the kernel version over 2.6.20 today. So the location of physical memory is fully transparent to the Linux kernel starting from 2.6.20.

On the other hand, to parse the physical memory relies the OS kernel knowledge. In our experiments, we selected a wide range of Linux distributions, including Ubuntu, Debian, CentOS and Fedora, with a variety of 13 different kernel versions, as shown in **Table 1**. Each kernel image is different because the source code of kernel is different. For example, the order and the offset of member in *task_struct*is different. Take offset of pid as an example, the offset is 0x4a8 in CentOS6.4, 0x2ac in Ubuntu12.04, 0x1e4 in Debian7.3.0, 0x2a4 in Fedora18. Any method to repair the semantic gap could not ignore these differences. Fortunately, VDSM uses *BackDriver* to hide these differences. *BackDriver* encapsulates properties and implementation details of corresponding object, opens the public interface to access OS semantic information to *FrontDriver*in the form of static library. During semantic view reconstruction, system only interact with *FrontDriver* to obtain the necessary kernel information. In other words, system just calls functions to accomplish reconstruction. For system user, the design of VDSM shields the differences of kernel versions, and thus enhances the versatility and portability.

6. Conclusion

We proposed VDSM, a lightweight and general approach for intrusion detection on virtualization platform. VDSM leverages driver separation mechanism and the key idea is to perform semantic view generation and OS semantics extraction with different modules. This modular architecture improves the efficiency of online processing as well as avoids semantic gap problem. We implemented VDSM and our experiments of the VDSM prototype on 13 Linux kernels demonstrate its effectiveness and portability. Moreover, our evaluation with 7 existing antivirus software further shows the performance efficiency of VDSM. While the main limitation of VDSM is that it assumes the hypervisor is absolutely safe. If hypervisor is infected every operation of VDSM becomes untrusted.

In future, we will develop our research on the security of hypervisor and enrich the functions of semantics extraction, such as system call and network connection, to achieve the fine-grained monitoring and to improve rootkit detection capability. Besides, based on VDSM we can also develop research on memory forensics. The hidden process binary code section we have captured can be translated into assembly code by reverse engineer and recorded as electronic evidence. Our goal is to create a security system that can monitor, pre-alert, and dispose of malware.

Acknowledgement

This work was supported by the National Natural Science Foundation of China(31371340), the Foundation of President of Hefei Institutes of Physical Science, Chinese Academy of Sciences (YZJJ201329), and the National Key Technologies Research and Development Program of China (No: 2016YFB0502604).

References

[1] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter and Radu Sion, "Sok: Introspections on trust and the semantic gap," in *Proc. of The 2014 IEEE Symposium on Security and Privacy*, pp.605-620, May 18-21,2014. Article (CrossRef Link).

[2] Pearce M, Zeadally S and Hunt R. "Virtualization: Issues, security threats, and solutions," *ACM Computing Surveys (CSUR)*, vol.45, no.17, pp.94-111, February, 2013. Article (CrossRef Link).

[3] Laniepce S, Lacoste M, Kassi-Lahlou M, et al., "Engineering intrusion prevention services for iaas clouds: The way of the hypervisor," in *Proc. of the 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pp.25-36, March 25-28, 2013. Article (CrossRef Link).

[4] Egele M, Scholte T, Kirda E, et al., "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol.44, no.6, pp.1-42, February, 2012. Article (CrossRef Link).

[5] Davis M,Bodmer S and Lemasters A,"HACKING EXPOSED MALWARE AND ROOTKITS," *McGraw-Hill Osborne Media*, 2009.

[6] McAfee Labs Threat Report,2015.Available: http://www.mcafee.com/cn/resources/reports/rp-quarterly- threat-q1-2015.pdf.

[7] Internet Security Threat Report, vol.20, 2015. Available:https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932GA-internet-security-threat-report-volume-20-2015-social v2.pdf.

[8] Vasilomanolakis E, Karuppayah S, Muhlhauser M and Fischer M, "Taxonomy and Survey of Collaborative Intrusion Detection," *ACM Computing Surveys*, vol.47, no.55, pp.55-88, July, 2015. Article (CrossRef Link).

[9] Kabiri P, Ghorbani A, "Research on Intrusion Detection and Response: A Survey," *International Journal of Network Security*, vol.1, no.2, pp.84-102, September, 2005.

[10] Garfinkel T, Rosenblum M., "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. of The Network & Distributed Systems Security Symposium*, pp.191-206, 2003.

[11] LKCD Linux Kernel Crash Dump[EB/OL]. Available:http://lkcd.sourceforge.net/.

[12] Jiang X, Wang X, Xu D, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proc. of The 14th ACM conference on Computer and communications security*, pp.128-138, 2007. Article (CrossRef Link).

[13] Pfoh J, Schneider C, Eckert C, "A formal model for virtual machine introspection," in *Proc. of The 1st ACM workshop on Virtual machine security*, pp.1-10, 2009. Article (CrossRef Link).

[14] Carbone M, Conover M, Montague B, et al., "Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection," *Research in Attacks, Intrusions, and Defenses*, vol.7462, pp.22-41, 2012. Article (CrossRef Link).

[15] Graziano M, Lanzi A, Balzarotti D, "Hypervisor memory forensics," in *Proc. of International Workshop on Recent Advances in Intrusion Detection*, vol.8145, pp.21-40, 2013. Article (CrossRef Link).

[16] Xiong H, Liu Z, Xu W, et al., "Libvmi: a library for bridging the semantic gap between guest OS and VMM," *Computer and Information Technology (CIT)*, in *Proc. of The IEEE 12th International Conference on IEEE*, pp.549-556, 2012. Article (CrossRef Link).

[17] Hay B, Nance K," Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no.3, pp.74-82, 2008. Article (CrossRef Link).

[18] Jones S T, Arpaci-Dusseau A C, Arpaci-Dusseau R H, "Antfarm: Tracking Processes in a Virtual Machine Environment," in *Proc. of The 2006 USENIX Annual Technical Conference*, pp.1-14, 2006. Article (CrossRef Link).

[19] Jones S T, Arpaci-Dusseau A C, Arpaci-Dusseau R H, "VMM-based hidden process detection and identification using Lycosid," in *Proc. of The fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 91-100, 2008. Article (CrossRef Link).

[20] Dolan-Gavitt B, Leek T, Zhivich M, et al.. "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. of The 2011 IEEE Symposium on Security and Privacy*, pp.297-312, May 22-25, 2011. Article (CrossRef Link).

[21] Fu Y, Lin Z., "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. of the 2012 IEEE Symposium on Security and Privacy*, pp.586-600, May 20-25, 2012. Article (CrossRef Link).
[22] ROBERT L. Linux Kernel Development,New York: Mac Millan Computer Publication, 2005.
[23] The Xen Project Power. [online] Available: http://www.xenproject.org/
[24] KVM. [online] Available: http://www.linux-kvm.org/page/Main Page
[25] QEMU. [online] Available: http://wiki.qemu.org/Main Page
[26] Cui C, Wu Y, Li P and Zhang X., "Narrowing the semantic gap in virtual machine introspection," vol.36, no.8, pp.31-37, 2015.
[27] Adore-ng. [online] Available: http://stealth.openwall.net/rootkits/
[28] KBeast. [online] Available: https://packetstormsecurity.com/files/108286/ipsecs-kbeast-v1.tar.gz
[29] Suterusu. [online] Available: https://github.com/dschuermann/suterusu
[30] Suneja S, Isci C, De Lara E, et al., "Exploring VM Introspection: Techniques and Trade-offs," *Acm Sigplan Notices*, vol. 50, no.7, pp.133-146, 2015. Article (CrossRef Link).

Chaoyuan cui born in 1972. PhD, associate professor. His main research direction includes system virtualization, architecture of cloud computing, information and communication security.

Yun Wu born in 1974. PhD, associate professor. Her main research direction includes system virtualization, architecture of cloud computing, data mining and machine learning.

Yonggang Li born in 1988, Ph.D. Candidate. His current research interests include operating system security, virtualization and cloud computing.

Bingyu Sun born in 1974. PhD, professor. His main research direction includes, data mining and machine learning.