J Supercomput
DOI 10.1007/s11227-011-0608-2

A VMM-based intrusion prevention system in cloud computing environment

Hai Jin · Guofu Xiang · Deqing Zou · Song Wu · Feng Zhao · Min Li · Weide Zheng

Abstract With the development of information technology, cloud computing becomes a new direction of grid computing. Cloud computing is user-centric, and provides end users with leasing service. Guaranteeing the security of user data needs careful consideration before cloud computing is widely applied in business. Virtualization provides a new approach to solve the traditional security problems and can be taken as the underlying infrastructure of cloud computing. In this paper, we propose an intrusion prevention system, VMFence, in a virtualization-based cloud computing environment, which is used to monitor network flow and file integrity in real time, and provide a network defense and file integrity protection as well. Due to the dynamicity of the virtual machine, the detection process varies with the state of the virtual machine. The state transition of the virtual machine is described via Definite Finite Automata (DFA). We have implemented VMFence on an open-source virtual machine monitor platform—Xen. The experimental results show our proposed method is effective and it brings acceptable overhead.

Keywords Grid computing · Cloud computing · Virtualization · Intrusion prevention · File integrity

1 Introduction

Grid computing [1] is an effective computing model to aggregate computing and storage resources in a distributed environment, and cloud computing [2] is regarded as

H. Jin · G. Xiang · D. Zou (✉) · S. Wu · F. Zhao · M. Li · W. Zheng
Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
e-mail: deqingzou@hust.edu.cn

H. Jin
e-mail: hjin@hust.edu.cn

Published online: 21 April 2011

a new direction of grid computing. Cloud computing is a new term for a long-held dream of computing as a utility, which has recently emerged as a commercial reality [1]. Foster et al. give the definition of cloud computing: A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the internet [2]. Cloud computing radically changes the usage mode of traditional hardware and software resources. There is no need for enterprises/organizations to purchase such expensive resources and spend much energy on configuration and maintenance of a system environment. Data center, a popular cloud computing application, possesses huge computing and storage resources, which gives cloud users the illusion of infinite computing and storage resources available on demand.

According to the statistical report [3] on the buzzword from Google, cloud computing becomes more popular than grid computing with the vigorous push of virtualization [4–6]. At the same time, the security of cloud computing brings us a large challenge due to its main applications in electronic commerce. The sensitive data and applications of end users are transferred to the cloud, so how to guarantee the security of such user information is a critical issue. Virtualization provides a new approach to solve the traditional security problems, and it also brings new security issues to computer systems [7]. The security of virtualization-based cloud computing come down to that of virtualization itself.

Virtualization is taken as the underlying infrastructure of cloud computing, and it can resolve certain security problems occurring during the evolution of cloud computing. The advantages of virtualization are described as follows:

(1) *Smaller Trusted Computing Base* (TCB): The code size of Virtual Machine Monitor (VMM) is far less than that of the traditional Operating System (OS). It means that VMM has less bugs and better robustness than the traditional OSes.
(2) *Better Isolation*: Virtualization provides better isolation than the traditional OSes. The applications in each Virtual Machine (VM) locate in a different address space on a single platform.

End users request for various services which are deployed in the cloud by the service providers. Services are deployed into different VMs separately, which are isolated from each other by VMM. Similar to a normal file, a VM can be easily migrated from one platform to another. The states of a virtual machine, such as *suspend*, *destroy*, *migrate*, and so on, vary with time. It is necessary to deploy traditional security tools with the consideration of the dynamicity of VM, such as the Intrusion Detection System [8] (IDS), firewall, File Integrity Monitoring Tool [9] (FIMT), and antivirus utilities.

In the traditional security area, the Intrusion Prevention System [10] (IPS) is one of important tools to detect and prevent illegal access. Intrusion detection and prevention are the two functions of IPS. Except for detecting network packets, file integrity monitoring is an effective approach to prevent the attacks from modifying the sensitive files.

In cloud computing environments, we propose a customizable defense system, called VMFence [11], deployed with distributed IPS and FIMT, which has the high

efficiency of detection and response. Detection processes, residing in the privileged VM, run in parallel, each of which detects the information flow from or to a VM. A detection process will be launched automatically when a new VM starts locally or is migrated from other hardware platforms. VMFence prevents malicious attacks in a virtualization based cloud computing environment.

The rest of this paper is organized as follows: Sect. 2 introduces the related work and background. In Sect. 3, we show the architecture of VMFence, and model the detection process through DFA. Section 4 presents the implementation of VMFence in detail. The evaluation of VMFence is described in Sect. 5. Finally, we give the conclusion and future work in Sect. 6.

2 Related work and background

In this section, virtualization is firstly introduced, and the basic mechanism of Xen [12–14] is explained in detail. Next, we present the related work about intrusion detection and file integrity monitoring.

2.1 Virtualization

The concept of virtualization was firstly introduced by IBM in the 1960s to provide concurrent, interactive access to a mainframe computer—IBM 360, which supports many instances of OSes running on the same hardware platform [4]. Virtualization technology, which can fully utilize the high performance of multicore [15], becomes a hot topic in both academia and industry recently. At the same time, virtualization gets support on the hardware layer from some manufacturers, such as Intel [16] and AMD [17]. Virtualization technology supports multiple OSes running on a single hardware platform, and provides a convenient means to manage the OSes. The OS and applications running on the virtualization management platform are considered as VMs. Normally, VMs are divided into two major types: process VMs and system VMs [5]. In this paper, we only discuss system VMs. The virtual machine monitor is the core component of system VM, which provides the abstract layer of underlying hardware for each OS running on it, such as Xen [12, 13], VMware [26], and Virtual PC [27].

Xen is a thin lay of system software on X86 architecture, and it locates on bare hardware to expose hardware abstraction slightly different from the underlying hardware [12, 14]. There are two different virtualization modes provided by Xen: full-virtualization and para-virtualization. Guest OSes run on Xen without any modification under full-virtualizaiton, while guest operating systems need to be modified under para-virtualizaiton for high performance. In order to utilize peripherals efficiently, the split device model is introduced and implemented in Xen architecture. Figure 1 shows the structure of split network driver and *blktap* architecture. The *frontend* (FE) and *backend* (BE) reside in isolated VMs, and communicate with each other by I/O ring and event channel mechanisms [14].

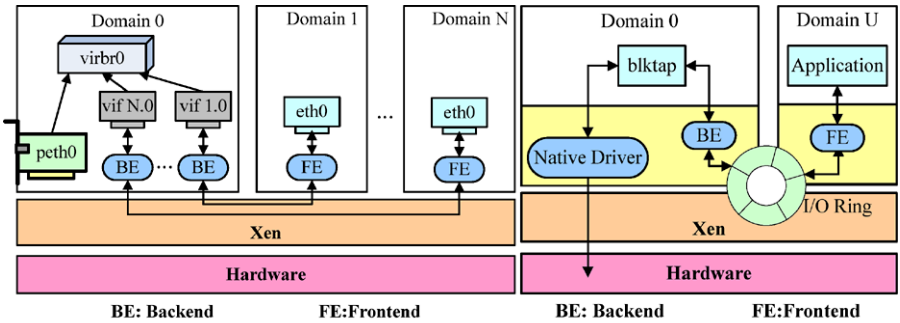


Fig. 1 The mechanisms of Xen

2.2 Intrusion detection

The concept of intrusion detection was first proposed by Anderson in 1980, which did not blossom until Denning published her seminal intrusion detection model in 1987 [8]. Intrusion detection involves determining that some entity, an intruder, has attempt to gain, or worse, has gained unauthorized access to the system. According to the source of detected data, IDS can be classified into Host-based IDS (HIDS) and Network-based IDS (NIDS). HIDS, such as OSSEC [28], collects the characteristics of system states, including the integrity of file systems, audit logs, network events, and the sequence of system calls. While NIDS, such as Snort [29], captures data packets by monitoring network traffic, it then analyzes the packets to make decisions whether they contain the malicious codes or not.

Dunlap et al. presented a system named ReVirt [18] that uses a virtual machine log and replay to analyze attacks including any nondeterministic events. ReVirt gives us a new direction to investigate intrusions as a result of reasonable time and space overhead. Garfinkel and Rosenblum firstly introduced the concept of the Virtual Machine Introspection (VMI) in Livewire [19]. Livewire is a new architecture for building IDS with the merits of both semantic-rich view and high resistance via VMI. IntroVirt [20] uses introspection and replay to improve security by detecting intrusions that occurred before the vulnerability was disclosed.

Kourai and Chiba have proposed a system called HyperSpector [21] which is a distributed, virtualization-based monitoring system in distributed computer systems. Using traditional multiple IDSes to protect distributed systems could lead to the increasing of unsafe points. HyperSpector has overcome such problem without any additional hardware by using virtualization.

2.3 File integrity monitoring

File integrity monitoring is an important component in HIDS, and it is used for the administrator to discover malicious behaviors. Tripwire [9] is an outstanding example of FIMT. There are four modes in Tripwire: *init*, *check*, *update*, and *test*. In *init* mode, the significant files are configured by the administrator, and it gives a snapshot of

these files. The administrator can verify whether these files are tampered in the *check* mode. Tripwire compares the current hash values of files with the previous values stored in database. Pennington et al. [22] proposed a storage-based intrusion detection system which allows the storage systems to watch for data modification characters.

I^3FS [23] intercepts file system calls and injects its integrity checking operations in the kernel mode. It performs checksum comparison in the critical path. XenFIT [24] is a file integrity monitor which is implemented on Xen. In XenFIT, the breakpoints are inserted in the monitored system, and intercept file system calls, for example, *open*, *close*, and *write*. It records the system call log, and sends it to the privileged VM. It is necessary to put an intercepting system call module in the monitored VM, which is easily disabled by the attackers.

3 The architecture of VMFence

First, we discuss an application scene of VMFence in cloud computing. Next, the architecture and components of VMFence are described in detail, which provide the function of intrusion prevention and file integrity monitoring in a cloud computing environment. At last, the state transition of detection process based on the state of the monitored VM is explained through Definite Finite Automata (DFA).

3.1 Application scene

Cloud computing is supported from a mass of high-performance hardware resources, and virtualization decouples the dependency between the underlying resources and system software.

Figure 2 shows an application scene in cloud computing environment, and there are four levels as follows:

(1) *Hardware platform layer*: The hardware platform in cloud computing environment consists of a few geographical distributed data centers connected by high speed network with each other; each of which has huge computing and storage resources.
(2) *Virtualization layer*: This layer is used to abstract and integrate the underlying resources as lots of independent entities, such as virtual machines or virtual clusters.
(3) *System platform layer*: Based on the interface provided by the virtualization layer, system software and application execution environment are configured and managed in this layer and can adapt to all kinds of hardware platforms.
(4) *Application layer*: It is unnecessary for the cloud users to know the accurate location of the resources. Various applications can be deployed and executed in the cloud, such as MPI program, ftp service, web service, and so on.

Except for these VMs providing services for cloud users, other privileged VMs are used to enhance the security of the cloud. The privileged VM just does control and management tasks, and has no user applications running on it. The reason is that the privileged VM is the security bottleneck of the whole platform. So, the less

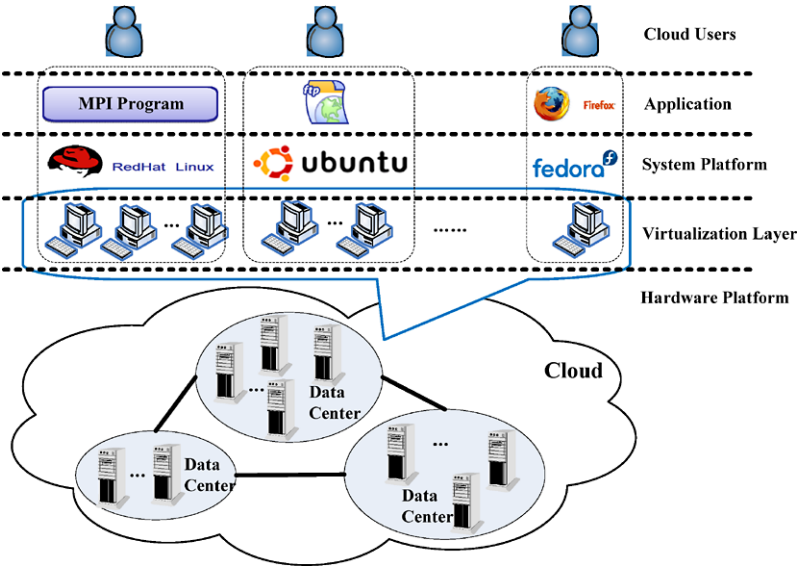


Fig. 2 An application scene in cloud computing environment

applications are installed in the privileged VM, the better is the security of the whole platform. A detection process executes in the privileged VM, which is responsible for the security of a VM. We mainly consider the state transition of a detection process in the privileged VM when the state of the corresponding VM changes. In this paper, the privileged VM and VMM are the TCB of the whole system, which is the same as other research projects [18–21].

3.2 The components of VMFence

VMFence is designed to provide flexibility and convenience to manage network packets detection and file integrity monitoring in cloud computing environment. Network intrusion detection and file integrity monitoring are the two functions of our design.

VMFence exploits the fact that the privileged VM is able to capture all network packets to or from other service VMs, thus the privileged VM has the ability to detect all packets without installing an instance of IDS in each service VM.

In the virtualization-based computing environment, the communication between service VMs must pass through the virtual bridge in the privileged VM, so the virtual bridge can be monitored by VMFence. The security can be assured by the isolation provided by VMM. When new VM starts or migrates from other node, or the existed service VM stops, the traditional network IDS cannot adapt itself to this dynamic situation. The IDS cannot utilize the multicore resource even if the underlying hardware holds huge computing power. When the network is busy, the drop rate will increase sharply.

In addition, file integrity monitoring is absolutely required for the security of user data. Traditional FIMT are deployed in the monitored system, which is probable to be tampered or masked by malicious attackers. Through the *blktap* architecture provided

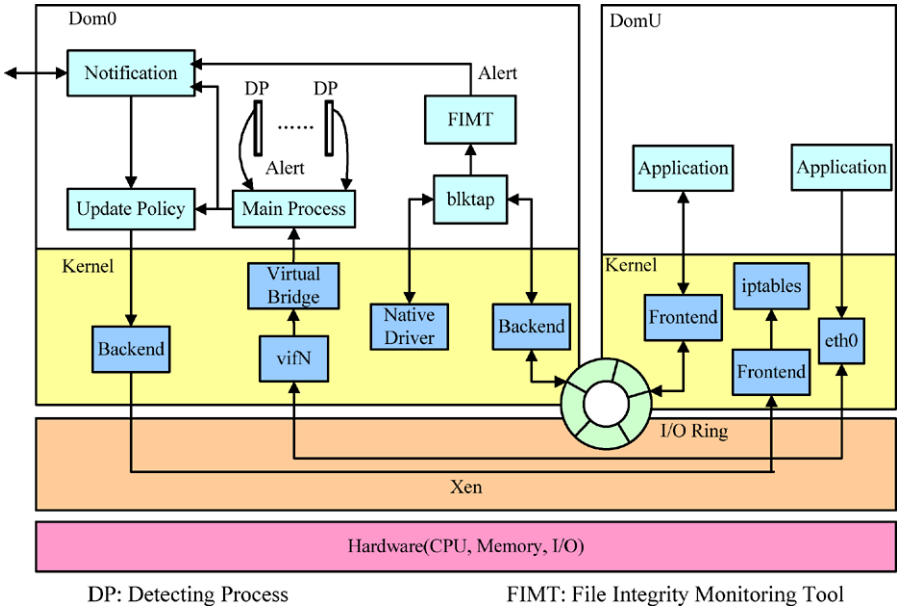


Fig. 3 The components of VMFence

by Xen, FIMT resides in the privileged VM. It is out of the catch of intruders, but can monitor file operations that occurred in the monitored VM.

We have proposed the architecture of VMFence in Fig. 3. The main process running in the privileged VM detects the VM created locally or migrated from other hosts. A new detection process starts with default or customized configuration. When there are some attacks, the main process will notify the administrator and send them to the backend node. According to the alert times during this period of time, new firewall rules will be created, and send it to the service VMs by the communication module. The *frontend* in service VMs listen to the *backend* and update local firewall policies. The detection process is paused if the corresponding VM is blocked because of waiting for I/O or suspending by the administrator, and it will continue the detection work if the VM runs again. When a VM is migrated or shut down by the administrator, the detection process for the VM will be killed by the main process at the same time. The states of the detection processes in VMFence vary with those of the corresponding VMs, and are adaptive to the state transition of VMs automatically.

FIMT in the privileged VM inspects the reading/writing operations in real time. When the sensitive files defined by cloud users are changed, alerts are generated to notify the cloud provider.

There are five components in VMFence:

(1) *Detection Component*: It has the main process to capture all network packets, and dispatches them to other detection processes according to its MAC address. Each detection process does a detection task according to the rules for the corresponding VM. There are some default rules for each service VM, and the administrator can customize these rules. When a new VM starts and the detection rules are

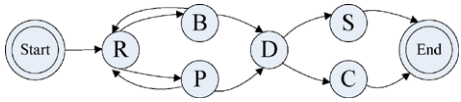


Fig. 4 The state transition of VM in Xen

configured for the VM, a detection process launches. When a VM migrates from another hardware platform, the configuration file will be transmitted at the same time, and the detection process will be started on the destination node.

(2) *Policy Updating Component*: This component is used for intrusion response. The main process collects all the alerts from detection processes, notifies the administrator, and sends them to the backend console in the distributed environment. Besides, firewall policies in a service VM must be updated if attacks on the VM are detected. If these measures do not work, the administrator can take other steps, such as pause or shutdown the attacked VMs.

(3) *Frontend and Backend Communication Component*: This component is designed to communicate between *frontend* and *backend*. When the policy updating component has created a new firewall policy for the service VM, the backend component must transfer this policy to *the* service VM and notify it. There is a frontend component in each service VM, which is responsible for allocating memory shared with the privileged VM. The frontend component will listen to the backend to determine whether there are updating firewall rules or not.

(4) *File Integrity Monitoring Component*: There is a great deal of service VMs for cloud users. They can customize the sensitive data of their own. FIMT is used to observe reading/writing operations in the backend. If the modified files belong to the protected set, the alerts are passed to the cloud provider through the notification component.

(5) *Notification Component*: It receives the service type and sensitive files defined by cloud users. At the same time, it collects the basic information about the VMs servicing for cloud users, and then sends it to them. Except for these, it collects alert information to notify the cloud provider.

3.3 The states of detection processes

The states of detection processes in the privileged VM vary with the transition of VM. The states of VM are relevant to the specific virtualization software. We take the case of Xen, and the state transition of VM in Xen is highlighted in Fig. 4. There are 6 states for a Xen VM:

- State *R* (Running): The domain is currently running on a CPU.
- State *B* (Blocked): The domain is blocked, usually is waiting for I/O information.
- State *P* (Paused): The domain is paused, usually because the administrator runs the command *xm pause*.
- State *D* (Dying): The domain is in process of dying, but has not completely been shutdown or crashed.
- State *S* (Shutdown): The domain is closed by the administrator.
- State *C* (Crashed): The domain is crashed because of a violent ending.

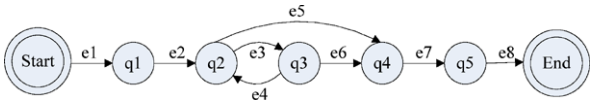


Fig. 5 The state transition of detection process

When the administrator runs *xm create*, a VM's state transfers to R. When the administrator runs *xm destroy* or *xm shutdown* on the VM, its state transfers to C or S.

This transition of detection process can be expressed by DFA in Fig. 5.

The DFA is described by a tuple $M = (Q, \sum, \delta, q_0, F)$ where $Q = \{Start, q_1, q_2, q_3, q_4, q_5, End\}$ is the finite set of states, $\sum = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$ is the input element, $\delta : Q \times \sum \rightarrow Q$ gives the set of transition, $q_0 \in Q$ is the initial state, *Start* in Fig. 5, and $F \subseteq Q$ is the final state, *End* in Fig. 5.

There are 7 states of a detection process. *Start* state is the initial state when VMM and control VM boot correctly. *End* state represents that this platform has been shut down. Other 5 states are depicted as follows:

- State q_1: a VM starts in local platform, or is migrated from another platform, but there is no detection process for the VM.
- State q_2: The detection process is running, $q_2 = \{R\}$.
- State q_3: The detection process is paused, because the VM is blocked or paused, $q_3 = \{B, P\}$.
- State q_4: VM is destroyed or migrated, while the detection process continues.
- State q_5: The detection process is killed, $q_5 = \{D, S, C\}$.

When the states of a VM change, the detection process for the VM will be adaptive to this transition. There are 8 events which trigger the transition between these states.

- Event e_1: The administrator creates a VM, but does not configure detection rules for the VM.
- Event e_2: The administrator configures detection rules for the VM.
- Event e_3: The domain is blocked because of I/O, or paused by the administrator.
- Event e_4: The domain's state changes to R.
- Event e_5: When the detection process is running, the VM is migrated to another platform, or destroyed by the administrator.
- Event e_6: When the detection process is paused, the VM is destroyed or migrated by the administrator.
- Event e_7: The detection process is killed by the main process.
- Event e_8: The whole VMM platform has been shut down by the administrator.

There are two unstable states: q_1 and q_4, because the relationship between a VM and a detection process is not always one-to-one mapping. The main process will explore this abnormity, and create or kill the detection process. States q_2, q_3, and q_5 are steady, and the states of the detection process will transform along with that of the corresponding VM. For example, the current state of a VM is q_1, and the main process will notify the administrator to configure rules for the VM. After the administrator configures detection rules for the VM (Event e_2), the corresponding detection process for the VM starts (State q_2). The state transition can be described as $\delta : q_1 \times e_2 \rightarrow q_2$. Figure 5 can reveal all situations of state transition.

In traditional IDSes, the detection will be terminated. In out prototype, however, state q_3 has been introduced to the system, which represents the detection process is paused corresponding to the VM's state. When a VM is in state q_2, event e_3 happens. The detection process will convert to state q_3 ($\delta : q_2 \times e_3 \rightarrow q_3$). After the VM continues (Event e_4), the detection process gets back to state q_2 ($\delta : q_3 \times e_4 \rightarrow q_2$). State q_3 reflects the relationship between a detection process and the corresponding monitored VM.

4 The implementation of VMFence

We build the experiment environment on Xen, an open-source VMM. But the methods mentioned in this paper can also be implemented on other virtualization platforms, such as VMware and Virtual PC. The reasons that we chose Xen are as follows:

- VMs running on Xen in the para-virtualization mode can have a similar performance as the operating systems running on native physical nodes.
- Xen supports almost all device drivers on Linux.
- Unmodified application binaries can run on guest OS.

A domain is the integration of guest OS and the applications running on it. The privileged VM on Xen takes responsibility for VM management, such as *create*, *destroy*, *pause*, *restore*, and we call it *Dom0*. Other service VMs in which services run are called *DomU* (Unprivileged Domain). All data in or from *DomU* must pass through *Dom0*, which plays the role of an agent, when *DomU* accesses the underlying hardware.

All *DomUs* run on the same machine, and they share the same physical resources. Xen provides an isolation mechanism for different *DomUs* which can share resources with each other if Xen permits. *Xenstore* is a storage system managed by *Dom0*. There is some configuration information in *Xenstore*, such as domain identification, domain name, and VM states. *Dom0* can get all such information, but *DomUs* can only see the information about itself for security.

4.1 Detection component

The detection component consists of one main watching process and some detection processes. The main process runs as a daemon taking charge of basic control and management tasks. The detection processes inspect the network flow through the VMs.

The first thing is to find the domain list running on this platform inspected by the main process. A kernel module, called *getdom*, in *Dom0* is registered as a character device, and the interface of this module is accessed by *getdom_read*, whose purpose is to read the list from *Xenstore* by *xenbus_directory*. Because we cannot use it in the user space, another function in the user space opens this character device and calls the function *read*.

When the main process gets the domain list, a data structure is defined to describe the domain. Figure 6 represents the domain structure in *Dom0*, and all the domain

Fig. 6 The domain structure

```
typedef struct_ Domain
{
   pid_t pid;                    // the process id
   char mac[6];                  // mac address of DomU
   int domain num;               // domain num
   char config_filename[100];    // configure file name
   struct _RuleList *rulelist    //point to rule list
   struct _Domain *next;         // point to next domain
} Domain;
```

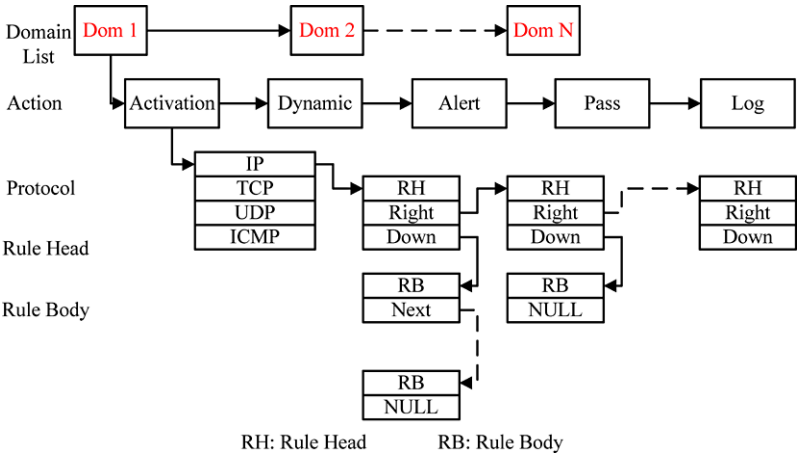


Fig. 7 The detection rule list

structures are organized as a list. In *Dom0*, the domain structure includes the domain identification of the detection process and the domain num. When a domain is created or migrated, the main watching process will insert a new item into the list. When a domain is migrated or stopped, the main watching process will delete it from this list. The main watching process carries out the updating operation on this domain list.

In VMFence, we adopt Snort as IDS, and Iptables as a firewall. Snort is an open-source IDS with high performance. The detection rules in Snort are organized as a three-level list: the actions (*Action*, *Dynamic*, *Alert*, *Pass*, *Log*), the protocol (*IP*, *TCP*, *UDP*, *ICMP*), and the rule head. Function *ProcessPacket* is the detect part in Snort. Packets captured from physical network will be dispatched to each detect process according to the domain's MAC address, and every detect process runs as an instance of *ProcessPacket*. Figure 7 shows the modification to the traditional rule list, a new classification of domains will be done before the action. Each detection process sends alerts to the main process when alerts appear, and the main process will collect these alerts by *select*.

The states of detection processes are controlled by the main watching process. When the service VM is blocked by I/O requests or paused (State q_3) by the administrator, the main process sends a signal, named *SIGSTOP*, to the corresponding

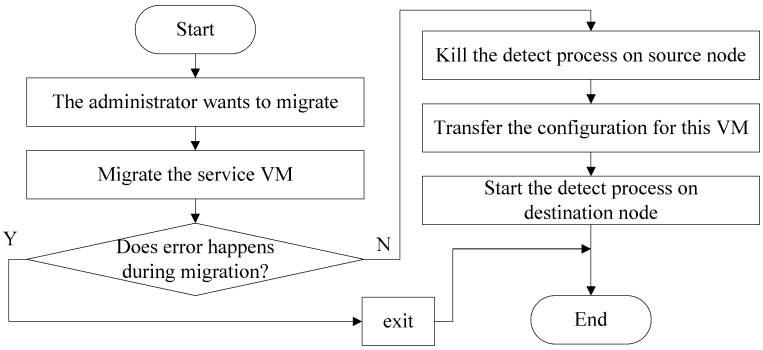


Fig. 8 The execution flow of the main process during migration

detection process. The detection process will be paused. After a service VM changes to the running state (State q_2), the main process sends the signal *SIGCONT* to the detection process. The detection process will continue to inspect network packets.

The migration process of VMFence varies from the implementation in Xen, since VMFence must adapt to the movement of VM. After the service VM is migrated successfully, the main process should reply to such change. The detection process is killed on the source node, and booted on the destination node. If detection rules have been configured for the VM, the configuration will be transferred before the detection process starts on the destination node. The whole process is highlighted in Fig. 8.

4.2 Policy updating component

After a detection process sends alerts to the main process, the main process gathers these alerts by *select*. The main process can get all the alerts on this platform from all detection processes, which is a benefit to do the global decision.

The main process can also get the information that some attacks have occurred on one VM, and get an overview of the whole system. Using this information, the main process makes decisions for each VM, and notifies the communication component between the *frontend* and *backend*. The communication component will take charge of updating firewall policies.

4.3 Frontend and backend communication component

When the policy updating component wants to update the firewall policies, it is essential to transfer data through the communication component between the *frontend* and *backend*.

The communication process between the *frontend* and *backend* is described in Fig. 9, and it can be described as follows:

- The *frontend* in *DomU* allocates memory (usually one page) in order to communicate with *Dom0*.
- The *frontend* grants the reference of shared pages to *Dom0* by *gnttab_grant_foreign_access_ref*.

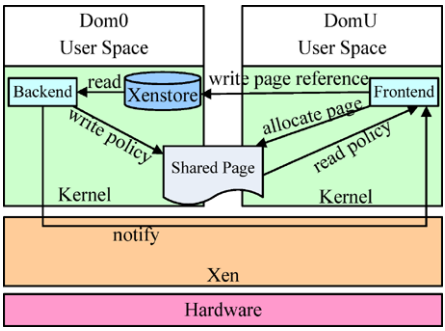


Fig. 9 The communication between frontend and backend

- The *frontend* writes the page reference of shared page to *Xenstore*.
- When new firewall policy has been created, the *backend* reads the shared page reference from *Xenstore*.
- The *backend* writes the new firewall policy file to this shared space.
- The *backend* notifies the *frontend* by the event channel.
- The *frontend* reads the policy from the shared page and updates local policies.

After all the above steps are accomplished, the Iptables of *DomU* will be updated. The policy updating process is implemented by the page sharing mechanism, and is faster than the traditional response by network.

4.4 File integrity monitoring component

File integrity monitoring is based on the *blktap* mechanism in Xen. *Blktap* is a driver in the user mode, and can directly manage disk activities with small performance overhead.

When a guest VM is started, a monitoring program running on the backend of the block device driver located in *Dom0* can observe file operations, and it parses the file operations occurring within the guest VM. In this paper, we will take the *ext2* file system for example. The super block and group descriptor shows the basic information about the file system. The sensitive file directory is defined for protection by cloud users, and it is detected in the backend. The data block numbers of the monitored files are copied to the memory and sorted in alphabetical order. When there is a read/write operation in the *frontend*, the corresponding data block is sent to the *blktap* driver. The data block number is parsed, and the monitoring program can determine if the file belongs to the monitored directory. If the data block is from the disk to the memory, it represents *read* operation. Otherwise, if there is new data block, it represents the *write* operation. In order to observe a *write* operation in real time, we set the disk flush time [25] to 0 by command *echo "0" > /proc/sys/vm/dirty_expire_centisecs*. The file integrity monitoring process is described in Fig. 10.

4.5 Notification component

Cloud users specify the service type and the directory/file which needs protection, and such information is transferred to the backend. Global policies, in regard to protecting

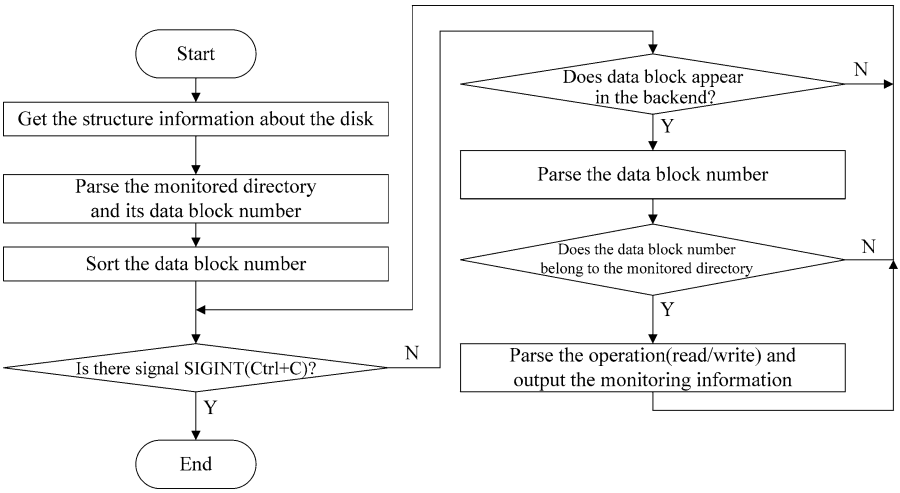


Fig. 10 The file integrity monitoring process

the objects, are made by cloud service providers. At the same time, the notification component gets the configuration and state information about the VMs deployed with services, and shows it to the cloud users.

On the other hand, the component collects all alert information related to VMs on a platform, and then all alert information is sent to the backend. After global policy is released by the providers, the component applies a new policy on this platform. The main function of this module is used for communicating with the backend.

5 Experiments and evaluation

In this section, we present the experiments which have been implemented on Xen 3.2, and then give our evaluation about VMFence.

5.1 The experiments

In our experiments, there are two nodes (*NodeA*, *NodeB*) and a backend node in the distributed environment. The hardware devices of *NodeA* and *NodeB* are two *Pentium* Quad-Core processors, 4 GB of memory, and Intel Gigabit Ethernet Controller. The frontend console is 2.0 GHz Pentium 4 processor, 512 MB of memory, and Realtek RTL8139 NIC.

Xen 3.2 and Fedora Core 8 have been preinstalled on each *NodeA* and *NodeB*. The backend console takes charge of collecting the alerts. There are 3 VMs running on *NodeA*: HTTPVM running http service, FTPVM running ftp service and SSHVM running SSH service at the initial state, and there is no service VM on *NodeB*. These service VMs have the same operating system as *Dom0*. During the execution, the three service VMs can migrate between *NodeA* and *NodeB*. Figure 11 shows the experiment environment in which we carry out our test.

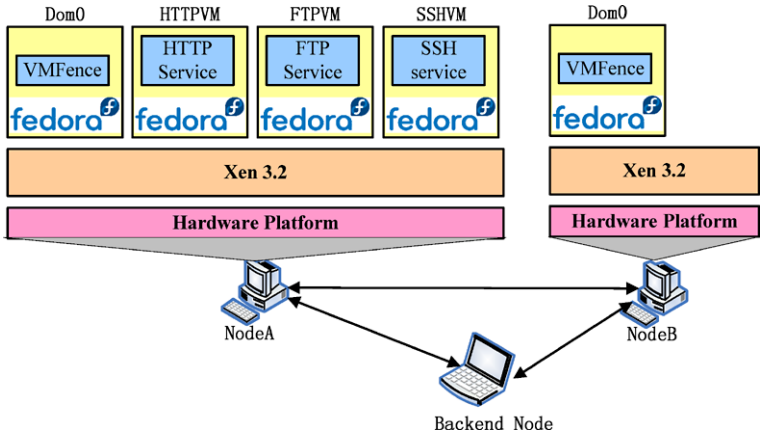


Fig. 11 The experiment environment

In order to simulate real network environment, we employ *DARPA98*, which is the intrusion detection dataset distributed by MIT Lincoln Laboratory. We rewrite the source address (*NodeB*) and destination address (*NodeA*) of network packets by *tcpprep*.

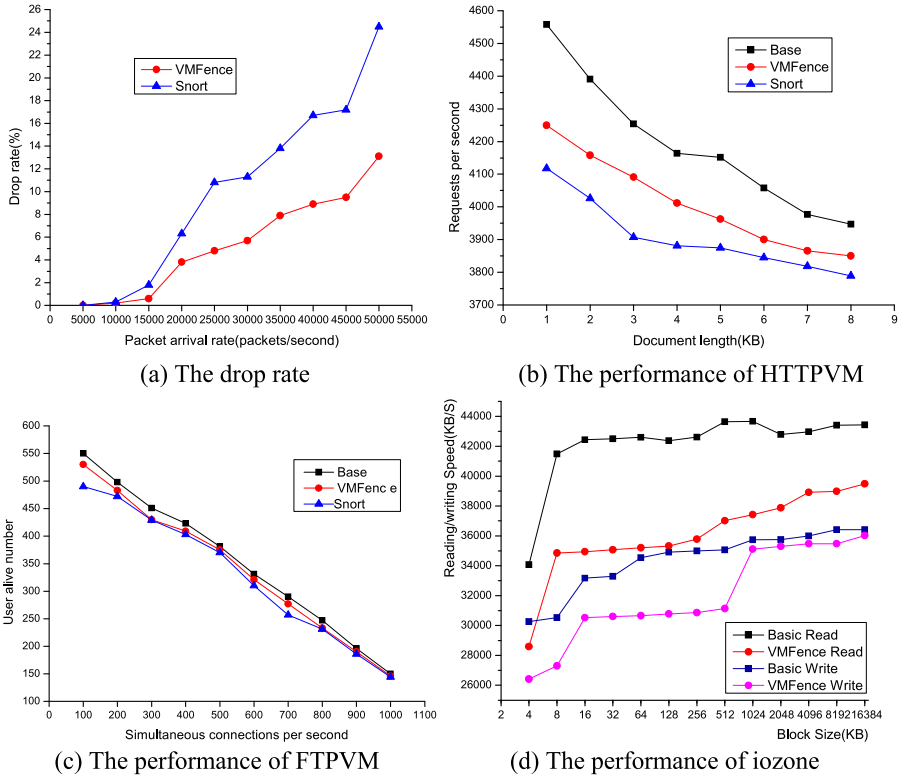
After three VMs has booted on *NodeA*, and configured with proper detection rules by the administrator, we replay *DARPA98* dataset on *NodeB*.

We test the drop rate, and compare the result with Snort. Snort is an excellent network intrusion detection system with low drop rate, but when the packet arrival rate is more than 45,000 packets per second, the drop rate increases sharply. VMFence, however, has a lower drop arrival rate, compared with traditional Snort. The drop rate about VMFence and Snort is depicted in Fig. 12(a). The reason is that the detect rules are configured for the services running in VMs, rather than all intrusion detection rules. And most important of all, the main process just captures all the packets in or from the virtual local network, and the detection work is done by other detection processes. This parallel structure has improved the efficiency of capturing network packets, and reduced the drop rate.

Then we measure the performance effect of VMFence. In the HTTPVM, *Apache* 2.2.6 is installed. We compare three results in different situations: (1) the base system without any detection (Base), (2) VMFence as the detect tool (VMFence), and (3) Snort as the detect tool (Snort). We take *Apachebench* as the testing tool of the performance of web service.

There are two arguments we specify: n represents the request number, c represents the concurrency degree. In our experiments, we choose $n = 10000$, $c = 100$. There are 10,000 requests in all, and 100 concurrent connections at one time.

With the increasing of document length, the requests per second decrease. The document length chosen by us is from 1 KB to 8 KB, and the requests per second are presented in Fig. 12(b). The requests per second on base system are more than other situations. Because there is no detect overhead, so the HTTPVM can deal with more web requests. These results show that VMFence has less overhead than traditional Snort, and it can respond to more requests than Snort. When the document length



(a) The drop rate

(b) The performance of HTTPVM

(c) The performance of FTPVM

(d) The performance of iozone

Fig. 12 The experiment results

increases, the requests per second drop gradually. From Fig. 12(b), we can find that the gap among these three vanishes gradually.

In the FTPVM, *vsftp* 2.0.5 has installed. The performance of FTPVM is evaluated by *dkftpbench*, which is an FTP benchmark program inspired by *SPEC web99*.

We compile the source code of *dkftpbench* and its dataset. The test file is copied to the public directory of anonymous user. We simulate 1,000 users fetching the default file (×1000k.dat) from FTPVM repeatedly, and stop after 20 seconds. Figure 12(c) shows the test results about FTPVM. When the simultaneous connections increase, the alive connections are made considerable reduction.

In order to evaluate the performance impact of file integrity monitoring, the reading/writing efficiency is tested by a well-known file system benchmark—*iozone*. The command we use is *iozone -a -s 512m -i 0 -i 1*. It indicates that only the efficiency of file reading and writing will be tested. The file size which we select is 512 MB, and it is twice as the memory size of the monitored system.

We focus on both file reading and writing operations, and Fig. 12(d) shows the speed of file reading and writing when the block size varies from 4 KB to 16384 KB. The horizontal axis represents the block size (KB), and the vertical axis represents the reading or writing speed (KB per second). As a whole, the writing performance is lower than the reading performance. When the block size increases from 4 KB

to 16384 KB, the overall efficiency improves gradually. According to all conditions from Fig. 12(d), we know that the reading and writing detection overhead is sustainable for most of systems. When the file block size is larger than 128 KB, the performance overhead brought by VMFence is less than 20%.

5.2 The evaluation of VMFence

In virtualization-based cloud computing environment, there are several service VMs running on the same hardware platform. Each service running in a specified VM has better isolation than traditional OS. How to enhance the security of these VMs is the primary issue we face to. VMFence is a VMM-based intrusion prevention system in cloud computing environment, and it can monitor network packets and file integrity in real time. The alert information are gathered and transferred to the backend, and the backend makes global decisions. The response channel is implemented through page-sharing mechanism, and quicker than network communication by which traditional IPS adopts.

The features of VMFence include:

1. *High adaptability*: VMFence supports configuring distinct policies for each service VM according to the style of services running on the VM. In the traditional operating system, the crash of one process may affect other processes, and even endanger the operating system itself. Virtualization provides better isolation than the process in the operating system, and we can deploy a single service in one VM to avoid interference from other services. These service VMs belong to different cloud users, and they are unable to interfere with each other. When the states of services VM change, the detection processes in the privileged VM adapt to this transformation automatically.
2. *High performance*: VMFence makes the detection processes in parallel to raise the detection efficiency. The privileged VM can capture all data from or to service VMs, and we start multiple detection processes to analysis the collected packets on multicore platforms. Otherwise, the file integrity monitoring brings little performance overhead to the whole system.
3. *Real time*: VMFence combines intrusion detection with intrusion prevention achieving real-time response when a crisis has been detected. New security rules will be created and applied automatically when an attack occurs frequently. On the other hand, the file operating information is output to the administrator in real time.

6 Conclusion and future work

The motivation of this paper is to design and implement VMFence, providing a customizable intrusion prevention system in avirtualization-based cloud computing environment. Cloud computing provides flexibility to utilize the underlying hardware resources adequately. VMFence supplies the cloud provider with a general management manner. The cloud provider can configure detection rules for each domain according to the type of service running in each VM. VMFence can dynamically adjust

detect processes by the number and the state of VMs, even VMs that migrate among different hardware platforms. In addition, the file modification information is collected in real time. We implement the VMFence by modeling the detection process and illustrate the state transition by DFA. The results of the experiment show that this method is useful for a virtualization-based cloud computing environment, especially for multicore CPU.

In the future, we will focus on building a general defense system on Xen, which includes IPS, the detection and behavior analysis of malware, and the secure risk evaluation of a virtual machine. In order to improve the security of a cloud computing environment, we will implement those through virtual machine introspection without modifying the VMM.

Acknowledgements The work is supported by National 973 Basic Research Program of China under grant No. 2007CB310900, National Natural Science Foundation of China under Grant No. 60973038, No. 60673174, and No. 60803114, National High-tech R&D Program (863 Program) under the Grant No. 2009AA01A402, Wuhan City Programs for Science and Technology Development under Grant No. 201010621211, Program for New Century Excellent Talents in University under Grant NCET-07-0334.

References

1. Foster I, Kesselman C, Tuecke S (2001) The anatomy of the grid: enabling scalable virtual organizations. Int J High Perform Comput Appl 15:200–222
2. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G (2009) Above the clouds: a Berkeley view of cloud computing. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley
3. Buyya R, Yeo CS, Venugopal S (2008) Market-oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. In: 10th IEEE international conference on high performance computing and communications. IEEE, Washington, pp 5–13
4. Rosenblum M, Garfinkel T (2005) Virtual machine monitors: current technology and future trends. IEEE Comput 38:39–47
5. Smith JE, Nair R (2005) The architecture of virtual machines. IEEE Comput 38:32–38
6. Adams K, Agesen O (2006) A comparison of software and hardware techniques for x86 virtualization. In: 12th international conference on architectural support for programming languages and operating systems. ACM, California, pp 2–13
7. Garfinkel T, Rosenblum M (2005) When virtual is harder than real: security challenges in virtual machine based computing environments. In: 10th workshop on hot topics in operating systems. IEEE, Santa Fe, pp 20–25
8. Machado RB, Boukerche A, Sobral JBM, Juca KRL, Notare MSMA (2005) A hybrid artificial immune and mobile agent intrusion detection based model for computer network operations. In: 19th IEEE international parallel and distributed processing symposium. IEEE, Denver, pp 191–198
9. Kim GH, Spafford EH (1994) The design and implementation of tripwire: a file system integrity checker. In: 2nd ACM conference on computer and communications security. ACM, Fairfax, pp 18–29
10. Chrun D, Cukier M, Sneeringer G (2008) Finding corrupted computers using imperfect intrusion prevention system event data. In: Computer safety reliability, and security, vol 5219, pp 221–234
11. Jin H, Xiang G, Zhao F, Zou D, Li M, Shi L (2009) VMFence: a customized intrusion prevention system in distributed virtual computing environment. In: 3rd international conference on ubiquitous information management and communication. ACM, Suwon
12. Barham P, Dragovic B, Fraser K, Harris SHT, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: 19th ACM symposium on operating systems principles. ACM, New York, pp 164–177
13. Pratt I, Fraser K, Hand S, Limpach C, Warfield A, Magenheimer D, Nakajima J, Mallick A (2005) Xen 3.0 and the art of virtualization. In: 2005 Linux symposium. USENIX, Ottawa, pp 65–77

14. Chisnall D (2007) The definite guide to the Xen hypervisor. Prentice Hall, New York
15. Gelsinger PP (2001) Microprocessors for the new millennium: challenges, opportunities, and new frontiers. In: 45th international solid state circuits conference. ACM, San Francisco, pp 22–35
16. Intel Staff. Intel 64 and IA-32 architectures software developer's manuals. Intel Corporation, November 2008
17. AMD Staff. AMD64 architecture programmer's manual. AMD Corporation, September 2007
18. Dunlap GW, King ST, Cinar S, Basrai M, Chen PM (2002) Revirt: enabling intrusion analysis through virtual machine logging and replay. In: 5th symposium on operating systems design and implementation. USENIX, Boston, pp 211–224
19. Garfinkel T, Rosenblum M (2003) A virtual machine introspection based architecture for intrusion detection. In: 10th network and distributed system symposium. IEEE, San Diego, pp 191–206
20. Joshi A, King ST, Dunlap GW, Chen PM (2005) Detecting past and present intrusions through vulnerability-specific predicates. In: 20th ACM symposium on operating systems principles. ACM, Brighton, pp 1–15
21. Kourai K, Chiba S (2005) HyperSpector: virtual distributed monitoring environments for secure intrusion detection. In: 1st ACM/USENIX international conference on virtual execution environments. ACM, Chicago, pp 197–207
22. Pennington AG, Strunk JD, Griffin JL, Soules CAN, Goodson GR, Ganger GR (2003) Storage-based intrusion detection: watching storage activity for suspicious behavior. In: 12th USENIX security symposium. USENIX, Washington, pp 1–15
23. Patil S, Kashyap A, Sivathanu G, Zadok E (2004) I^3FS: an in-kernel integrity checker and intrusion detection file system. In: 18th USENIX large installation system administration conference. USENIX, Atlanta, pp 67–78
24. Quynh NA, Takefuji Y (2007) A novel approach for a file-system integrity monitor tool of Xen virtual machine. In: 2nd ACM symposium on information, computer and communications security. ACM, Singapore, pp 194–203
25. Bovet DP, Cesati M (2005) Understanding the Linux kernel, 3rd edn. O'Reilly, Sebastopol
26. VMware Home Page. http://www.vmware.com
27. Virtual PC Home Page. http://www.microsoft.com/windows/virtual-pc
28. OSSEC Home Page. http://www.ossec.net
29. Snort Home Page. http://www.snort.org