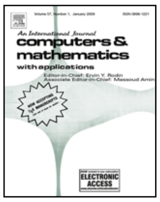
Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa



A guest-transparent file integrity monitoring method in virtualization environment

Hai Jin [*], Guofu Xiang, Deqing Zou, Feng Zhao, Min Li, Chen Yu

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

ARTICLE INFO

Keywords:
File integrity protection
Transparent monitoring
Real time
Xen

ABSTRACT

The file system becomes the usual target of malicious attacks because it contains lots of sensitive data, such as executable programs, configuration and authorization information. File integrity monitoring is an effective approach to discover aggressive behavior by detecting modification actions on these sensitive files. Traditional real-time integrity monitoring tools, which insert hooks into the OS kernel, are easily controlled and disabled by malicious software. Such existing methods, which insert kernel module into OS, are hard to be compatible because of the diversity of OS. In this paper, we present a non-intrusive real-time file integrity monitoring method in virtual machine-based computing environment, which is transparent to the monitored system. The monitor is isolated from the monitored system, since it observes the state of the monitored system from the outside. This method brings two benefits: detecting file operations in real time and being invisible to malicious attackers in the monitored system. Furthermore, a kind of file classification algorithm based on file security level is proposed to improve efficiency in this paper. The proposed file integrity monitoring method is implemented in the full-virtualization mode supported by the *Xen* platform. The experimental results show that the method is effective with acceptable overhead.

1. Introduction

One of the fundamental goals of computer security is to ensure the integrity of system resources [1]. A large number of safe accidents result from critical files modified in the system. The attackers intrude into the computer system, and hide their traces by tampering critical files, such as executable files and system logs. File Integrity Monitoring (FIM) is one of the most popular approaches to observe hostile behaviors, such as modifying system log, appending backdoors, inserting Trojan horses.

In order to avoid serious damage to the system, it is necessary to observe malicious behaviors in the system. Current FIM can be classified into two types: Periodic FIM (PFIM) and Real-time FIM (RFIM). PFIM tools compare current attributes of the files with previously gathered, for example, the owner, the content, and the last modification time. *Tripwire* [1] gives a snapshot of the protected files firstly, and the administrator can verify their integrity periodically. Trusted Platform Module (TPM) provides integrity authentication for a trusted path (BIOS, boot sector, OS kernel, applications) during the starting procedure of the system. It is difficult to be compromised by attackers because the trust chain is extended from the hardware. PFIM verifies the integrity of the system at one "point", rather than during the whole "process". RFIM tool works in the OS kernel to intercept file operations during the execution process of the system. *XenFIT* [2] is a real-time integrity monitor

[*] Corresponding author.
E-mail address: hjin@hust.edu.cn (H. Jin).

0898-1221/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2010.01.007

on the well-known *Xen* virtualization platform [3–5]. It puts several hooks into the kernel of the monitored system, and intercepts system calls related with these file operations. In order to monitor file operation in real time, a kernel module must be inserted in the monitored system. However, this module is easily attacked or masked by rootkits.

So, perfect FIM tool should satisfy the following requirements:

(1) *Attack resistant*: FIM tool should continue working even if the monitored system is completely controlled by malicious software (malware). In *XenFIT*, the system call interception module runs in the monitored system, it may be disabled by malware.
(2) *Real time*: FIM tool should get enough information in real time when configuration or executable files are modified by malicious intruders. *Tripwire* can check the integrity of files periodically, and TPM can measure any file during the system boots, but they cannot ensure the file security during the entire runtime of the system. Furthermore, the administrator does not know when the system begins to be unreliable, and which users tamper critical files.
(3) *Transparence*: FIM tool should neither modify the monitored system, nor insert any kernel module. *XenFIT* inserts multiple breakpoints in the monitored system, which restricts the practicability of the monitor. There are various monitored systems, for example, different types of OSes, diverse versions of the same class.

Virtualization technology gives people a novel approach to meet these requirements mentioned above. It provides the isolation between FIM tools and malwares, and an advanced FIM design based on such technology is feasible. From the report of IDC [6], virtualization [7–9] becomes an integral part of the IT infrastructure, and the virtualization services market will grow sharply in the future 5 years. Virtualization decouples OS with the underlying hardware, and supports multiple OS instances on the single hardware platform. An OS instance and its upper applications are encapsulated as a Virtual Machine [8](VM). The core component of virtualization is Virtual Machine Monitor [9](VMM), which is a "thin" software layer located under the traditional OS. The VMM owns full domination of the underlying hardware, and it can monitor all events happened in the VM. At present, the design tendency of security systems in virtualization environment is to pull the monitor tool out of the Monitored VM (MVM) and deploy it in another protected VM. Under the virtualization architecture, FIM tools need to observe the file changes on the monitored VM, and prevent possible attacks from the VM at the same time. The monitoring points are moved from the monitored system to the VMM, and cannot be sensed by the monitored system.

In this paper, we present a guest-transparent RFIM method in virtual computing environment. Intercepting file operations is implemented in the VMM, and it is no need to append any module in the monitored system. The RFIM tool locates in another Privileged VM (PVM), and observes file operations in the MVM. We can obtain enough file information, including process identity, file name, file operation, time et al., when one critical file is modified in the MVM. The RFIM tool is isolated from the MVM and transparent to the MVM. All files are classified into three categories based on the file security level. The file classification algorithm is proposed to take different actions on the files which belong to different sets.

The rest of this paper is organized as follows: Section 2 introduces the related work and background. Section 3 proposes the RFIM method in the virtual computing environment and discuss file classification according to the significance level. Section 4 describes the implementation of RFIM in an intrusion prevention system, named *VMFence* [10], based on the *Xen* platform. The experiments on *Xen* are described in Section 5. Conclusions and future work are presented in Section 6.

2. Related work

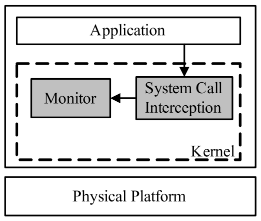
File integrity monitoring is one function of Host-based Intrusion Detection System [11] (HIDS), and it is used for the administrator to discover malicious behaviors. In this section, we introduce PFIM and RFIM respectively.

Tripwire [1] is a representative example of PFIM. There are four modes in *Tripwire*: *init*, *check*, *update*, and *test*. In *init* mode, the significant files are specified by the administrator, and *Tripwire* gives a snapshot of these files. The administrator can verify whether these files are tampered in *check* mode. *Tripwire* compares the current hash values of files with the previous values stored in a specified database. The problem of *Tripwire* is the selection of inspection cycle. If the cycle is too short, frequent inspection will impact on the system performance, otherwise, it cannot detect the variation in time. In addition, *Tripwire* executes on the monitored system, and is easily disabled by intruders with the administrator privilege. Except for *Tripwire*, *AIDE* and *Samhain* are the similar tools. Pennington et al. [12] proposed a storage-based intrusion detection system which allows the storage systems to watch data modification. This project implements file integrity monitoring when file modification happens on a file server in the distributed environment.

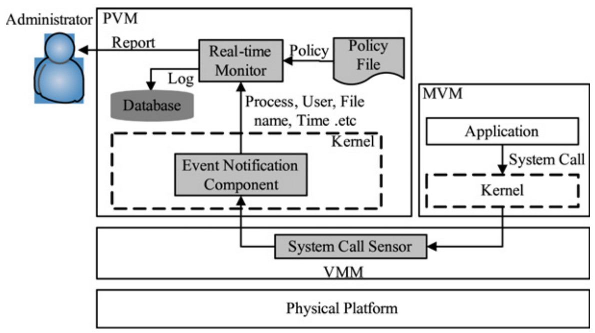
I^3FS [13] intercepts system calls and injects its integrity checking operations in the kernel mode. It performs checksum comparison in the critical path. *XenFIT* is a file integrity monitor which is implemented on *Xen*. The breakpoints are inserted in the monitored system, and intercept these system calls about file operations, for example, *open*, *close*, and *write*. It records the system call log, and sends it to the PVM. However, It is necessary to put an intercepting system call module in the MVM.

With the emergence of multi-core processor, virtualization technology is widely applied to various aspects of computer systems. VM-based security [14,15] becomes important increasingly. The isolation provided by VMM enhances the security of the detector, while brings a new problem, semantic gap simultaneously.

Virtual Machine Introspection [16,17] (VMI) was firstly proposed by Tal Garfinkel and Mendel Rosenblum in the project, *Livewire* [16], which is a new architecture for building an intrusion detection system with the merits of both high resistance and excellent visibility via VMI. VMI is an effective method to enhance system security including intrusion detection [18], malware detection [19,20], and honeypot [21–23]. *VMwatcher* [24] exports the resources in the MVM to the trusted host



(a) The traditional RFIM method.



(b) The proposed RFIM method.

Fig. 1. A comparison between traditional RFIM method and the proposed RFIM method.

environment, and carries out malware detection through anti-virus tools deployed on the host. It has implemented out-of-box detection and avoided these anti-virus tools under attack. Observing the sequences of system calls in a VM, and watching the behavior of processes are the remarkable characters of *VMscope* [25]. It provides a novel means to analyze malicious processes through honeypot. *Lares* [26] is an architecture for secure active monitoring using virtualization. The hooks are deployed in a untrusted guest VM, and they are protected by the VMM. *Xenaccess* [27] is an open source VMI library on *Xen*. It provides a convenient interface for programmers by encapsulating *libxc* and *libblktap*. It is a library to observe the states of another VM, but the functions must be implemented by users. On February 2008, *VMware* has released a VMI library, *VMsafe*, which provides an application program interface for other security manufacturers to improve the security of the VMM, for example, *Symantec*, *McAfee* etc.

For current FIM methods, the monitoring tools are easily destroyed by malicious attacks. On the other hand, current VM-based security systems do not provide the FIM function. Our paper aims to design and implement a real-time and guest-transparent FIM approach using virtualization technology.

3. Real-time integrity monitoring

3.1. Guest-transparent real-time integrity monitoring architecture

For most of the rootkits, they are installed into the system, and modify the critical file (*/usr/bin/{ls, ps}*) to hide their trace. Current monitoring methods intercept these system calls in the target system, and the rootkits can mask them without difficulty. However, the approach proposed in this paper puts the monitoring point in the VMM layer, which is isolated from the attacker. It has the real view about the target system with the function of semantic reconstruction.

Fig. 1 shows the difference between traditional RFIM method and our method. In order to monitor file modification, traditional RFIM tool is deployed as a kernel module. It inserts several hooks to intercept *read* and *write* system calls. That is, the traditional RFIM tool resides on the monitored system. While in our method, the monitor locates in a privileged VM, which is out of the range of intruder. There is no need to insert any additional module into the monitored VM, which is completely transparent to attackers.

An obstacle for VM monitoring is the semantic gap between applications in MVM and VMM. Semantic reconstruction is introduced to eliminate the gap. Current VMI library is inadequate for file integrity analysis. For example, *Xenaccess* tracks four operations: file creation and deletion, and directory creation and deletion. In order to obtain detailed file information, the system call sensor must be deployed in the VMM, which is a minor modification to the monitor and can help to get total information such as user identity, process identity, file name, operation time etc.

For our non-intrusive method, three basic components located at different levels are described as follows:

(1) *System call sensor in the VMM*: In order to monitor file operations in real time, this module captures system calls about file operations, such as *create*, and *write*. In the VMM level, it monitors all events occurred in each VM. When an application in the MVM invokes a system call, the execution flow is trapped into the VMM. This module parses system call parameters about file operations. After that, it will inform the event notification module in the PVM.
(2) *Event notification in kernel mode*: It transfers system call parameters and provides the interface for the monitor in the PVM. It is a bridge between the system call sensor and the monitor.
(3) *Real-time monitor in user mode*: This module implements the key function of RFIM. It obtains parameters and responds to file modification, when system calls are acting on the protected file set. According to the system traces, files are classified into three types: *significant*, *sensitive* and *ordinary*.

```
<monitored_file>
    <significant_file>
        <name>/bin/ls</name>
        <name>/bin/ps</name>
        <name>/bin/netstat</name>
    </significant_file>
    <sensitive_file>
        <name>/bin/sh</name>
        <name>/sbin/init</name>
    </sensitive_file>
</monitored_file>
```

(a) The format of specified files.

Rank	Count	Frequency	Command
1	519	23.6447%	ls
2	356	16.67422%	cat
3	287	13.0752%	cd
4	137	6.24146%	vim
5	69	3.14351%	sh
6	35	1.59453%	ifconfig
7	31	1.4123%	make
……			
106	1	0.0455581%	diff

(b) The usage frequency of commands.

Fig. 2. The input of file classification algorithm.

3.2. File classification algorithm

In traditional OS, file attributes contains distinct privilege of different users (owner, group, and other), such as readable (r), writable (w) and executable (e), which can prevent unauthentic access in a way. But this permission is invalid to the administrator, *root* in *Linux*. The administrator can access all files without any limitation. Therefore, if an attacker steals the administrator privilege by buffer overflow, he can cause serious damage on the target file. Normally, files have various security level in system. In order to quantify file security level, the weight of file i is defined as follows:

$$w_i = \alpha * f_i + \beta * d_i \quad (\alpha + \beta = 1). \tag{1}$$

w_i represents the weighted value of file i, f_i denotes the access frequency of file i, and d_i illuminates the significance of the directory which file i belongs to. For example, the directory for */usr/sbin/useradd* is */usr/sbin*, rather than */usr*. The larger w_i is, the more important the file is. The tuning variables, α and β, represent the proportion of the frequency and the significance of directory respectively and the sum of α and β is equal to 1.

$$\text{file } i \in \begin{cases} s_{\text{sig}} & \text{if } w \in [t_{\text{sig}}, 1] \\ s_{\text{sen}} & \text{if } w \in [t_{\text{sen}}, t_{\text{sig}}) \\ s_{\text{ord}} & \text{if } w \in [0, t_{\text{sen}}). \end{cases} \tag{2}$$

In this paper, we define two threshold values for *significant* file and *sensitive* file: t_{sig} and t_{sen}. If w_i is larger than the threshold values, file i belongs to the corresponding set (s_{sig} or s_{sen}). Otherwise, file i belongs to s_{ord}. The set which file i belongs to is defined as follows:

(1) *Significant* file: The user can read or execute this file, but it cannot be modified by any users, including *root* in *Linux*. In other words, the file is modification-forbidden for all users after system installation in the MVM. In *Linux*, some executable files are used to display the system state, such as processes (*/bin/ps*), files (*/bin/ls*), network connections (*/bin/netstat*). If these executable files have been tampered or replaced, the intruder can hide their malicious behavior. In the case of defenders, it is hard for them to discover the intruder's trace. When significant files are modified in the MVM, the monitor in the PVM should detect this malicious action and prevent any damage on the files.

(2) *Sensitive* file: The user can modify this file, but the operations must be recorded in the system log. In *Linux*, all configuration files locate under the directory */etc*. For example, */etc/profile* defines the *shell* environment after the users login. Any modification on software configuration may influence the system behavior. The intruders, who expect to gain private information, should destroy the system by modifying this file. It is necessary to log all things happened in the MVM for future analysis. The log recorder is isolated with the MVM to enhance the security of the monitor.

(3) *Ordinary* file: Other files except for *significant* and *sensitive* files, including temporary or inessential files. All file operations on these files are frequent, and the monitor does not need to monitor these files. For example, directory */tmp* in *Linux* keeps the temporary data during process execution. All operations on these files are ignored, and are not recorded in the log file.

The target of file classification is to take different actions for files which belong to different sets. All files in s_{sig} are write-forbidden for all users. All operations on the files in s_{sen} are logged in the PVM. The administrator should define the minimal s_{sig} and s_{sen} for the performance consideration. The more the files need to be protected, the longer the monitor spends. The administrator can initialize the file sets at the beginning, and some files are appended into these sets according to the command log. d_{sig} represents the set of significant directories, and d_{sen} represents the set of sensitive directories. They are defined by the administrator. File classification algorithm demonstrates how to determine s_{sig} and s_{sen}. At the beginning, set s_{sig} and s_{sen} are empty. The administrator defines *significant* files and *sensitive* files, and appends them to the above two sets. According to the file access frequency, the weighted values of the above files are calculated. When the weighted value w_i of an executable file exceeds the threshold value, t_{sig} or t_{sen}, it belongs to the corresponding set, s_{sig} or s_{sen}.

Algorithm 1: File classification algorithm
Input: User's command log, policy file, α, β, d_{sig}, d_{sen}, t_{sig}, t_{sen}
Output: s_{sig}, s_{sen}

```
procedure FileClassification
    $s_{sig}$ and $s_{sen}$ are empty
    read the policy file specified by the administrator
    append the specified file to $s_{sig}$ or $s_{sen}$
    get the frequency of each command ($f_i$),
    the total of non-repetitive commands (count)
    for ($i = 0$; $i < count$; $i$++)
    {
        if (file $i$ is not specified by the administrator)
        {
            if (file $i \in d_{sig}$)
                $d_i = 1$
            else if (file $i \in d_{sen}$ )
                $d_i = 0.5$
            else
                $d_i = 0$
            $w_i = \alpha * f_i + \beta * d_i$
            if ($w_i \geq t_{sig}$)
                append $f_i$ to $s_{sig}$
            else if ($w_i \geq t_{sen}$)
                append $f_i$ to $s_{sen}$
        }
    }
end procedure
```

4. RFIM implementation in VMFence

In this section, we firstly introduce a network-based intrusion prevention system in virtual computing environment, named *VMFence*. *VMFence* is a network-based intrusion prevention system in the distributed virtual computing environment, implemented in our previous work. Multiple VMs with different services run on the same platform. The PVM can monitor all network flow from or to the MVM, consequently all detections can be done in the PVM. Due to different requirements of service type and file security level, the administrator can configure detection rules for each VM. The detection components for all VMs can run in parallel to improve the detection efficiency, especially on the multi-core platform. When the MVM is migrated to another physical node, the detection rules must be transferred to the destination node. In addition, the detection process reboots on the destination platform according to these detection rules.

In *VMFence*, RFIM method aims to defend malicious modifications on the file in real time. In order to implement the RFIM method, the environment must satisfy the following two requirements:

(1) The guest VM is unmodified, which provides the transparence to the applications in the MVM. There is no need to insert any module in the kernel of the MVM.
(2) System calls can be intercepted in the VMM in real time. In order to implement the RFIM method, all system calls must be sensed in the VMM layer, and only these relevant to file operations are selected for further analysis.

The reason why we choose *Xen* as the VMM is for stability and high efficiency. We just take *Xen* for an example, and the same idea can be implemented on other virtualization platforms, such as *VMware*, *QEMU* [28]. On the *Xen* platform, a VM is called one domain. The privileged domain is called *Domain0*, which controls and manages other unprivileged domains named *DomainU*. *Xen* supports two different VM types: para-virtualization and full-virtualization. Guest operating systems need to be modified under para-virtualization for high performance, while guest operating systems run on *Xen* without any modification under full-virtualization with the help of hardware virtualization extension. Full-virtualization *DomainU* must get support from CPU, such as Intel VT [29,30] or AMD SVM [31].

4.1. Definition of file set

The file sets (s_{sig} and s_{sen}) are determined by the administrator and the command log. The administrator specifies the file sets in the form of XML. Fig. 2(a) gives an example of the file sets which are specified by the administrator. From this figure, the two sets are initialized by the administrator as $s_{\text{sig}} = \{/bin/ls, /bin/ps, /bin/netstat\}$, $s_{\text{sen}} = \{/bins/sh, /sbin/init\}$.

In addition, files can be appended to the sets dynamically. If file *i* is not specified by the administrator, its weighted value is calculated according to formula (1). The weighted value of file *i* is decided by its usage frequency and the directory which it belongs to. We take the above calculation approach on *Linux* as an example. Without loss of generality, α and β are set to 0.5, which indicate the two parts have the same significance.

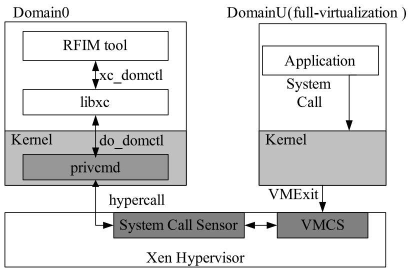


Fig. 3. The execution flow of RFIM.

First of all, we must get the frequency of each command for certain user. Considering *root* in *Linux* as an example, the command history which the user invokes is stored in *.bash_profile* under the user's home directory. The usage frequency of each command is acquired by the underlying *shell* code: *history* | *awk* '{*CMD* [\$2]++; *count*++;} *END*{*for* (*a in CMD*) *print CMD*[*a*] " " *CMD*[*a*]/*count* ∗ 100 "%" *a*;}' | *grep*-*v* "./" | *column* -*c*3 -*s*" " -*t* | *sort* -*nr* | *nl*. We record the command usage in one day. Fig. 2(b) outputs the statistical results of each command.

If file i belongs to $d_{\text{sig}} = \{/sbin, /usr/sbin, /usr/local/sbin\}$, we set $d_i = 1$. The commands executed on files in d_{sig} by the administrator can change the system's state. And if file i belongs to $d_{\text{sen}} = \{/bin, /usr/bin, /usr/local/bin\}$, we set $d_i = 0.5$. The commands can be executed on files in d_{sen} by any user. If file i belongs to none of them, we set $d_i = 0$. As *ls* is specified by the administrator, it is no need to calculate its weighted value. The weighted value of *cat* is calculated as follows: *cat* belongs to the directory */bin* (d_{sen}), and $d_i = 0.5$, and the usage frequency of *cat* is 0.1667, so the weighted value $w_{cat} = 0.5 * 0.1667 + 0.5 * 0.5 = 0.33$. The calculation for other files are similar to this.

According to the discussion above, we set the threshold values of s_{sig} and s_{sen}: $t_{\text{sig}} = 0.3$, and $t_{\text{sen}} = 0.25$. At last, these sets are: $s_{\text{sig}} = \{/bin/ls, /bin/ps, /bin/netstat, /bin/cat, /sbin/ifconfig\}$, $s_{\text{sen}} = \{/bin/sh, /sbin/init, /usr/bin/vim, /usr/bin/make\}$. The other files in the system belong to s_{ord}.

4.2. Guest-transparent monitoring using hardware virtualization

Based on hardware virtualization, our prototype is transparent to the MVM. Although hardware virtualization is related to CPU type, there are few differences between Intel VT and AMD SVM, especially the instructions about virtualization and their usage. We select Intel VT due to its broad market and detailed documentation.

Intel CPU with VT, which supports instructions about virtualization [30] (*VMXON*, *VMLAUNCH* etc.), provides two processor modes: *root* mode and *non-root* mode. *Xen* runs in *root* mode, and each full-virtualization VM runs in *non-root* mode. When special events (such as privilege instructions, external interrupts, and page faults etc.) happen in a full-virtualization VM, they will trigger the transition from *non-root* mode to *root* mode (*VMExit*). The states of guest OS (registers and context) within the VM are stored in the *VMCS* region, which is maintained by the VMM. After that, the VMM judges the reasons of *VMExit*, and executes the corresponding handler. After the VMM handles with the events, CPU resumes the execution flow of Guest OS. At the same time, the processor transforms from *root* mode to *non-root* mode (*VMEntry*).

In *Linux*, system call is the interface between user mode (ring 3) and kernel mode (ring 0), and there are two different implementation methods of system calls. The first one is through software interrupt (*int* \$0*x*80) [32]. When one process in user mode invokes a system call, the kernel checks its privilege and input parameters. The transition process from user mode to kernel mode is time-consuming. Therefore, fast system call (*sysenter*) provides a quick transition method from user mode to kernel mode. Because there is no privilege inspection and stack operation, its execution speed is faster than *int* \$0*x*80. At present, fast system call is applied in *Linux* 2.6. The related registers (*SYSENTER_CS_MSR*, *SYSENTER_EIP_MSR*, *SYSENTER_ESP_MSR*) must be prepared before instruction *sysenter* executes. Register *SYSENTER_EIP_MSR* indicates the entry point of function *sysenter_entry*.

Fig. 3 explains the relationship among function calls in RFIM. Library *libxc* is a user-space (ring 3) interface which invokes *hypercall*. In the implementation of RFIM, we operate on library *libxc*. The value of *SYSENTER_EIP_MSR* is modified when the monitor runs. The basic mechanism of system call interception is to trigger page fault at each time of system call invocation. When an application invokes a system call, page fault happens if the entry address does not exist. Page faults lead to "trap" into the VMM, and the system call sensor can implement system call interception. The above method will increase the usage frequency of *VMExit*, which is the overhead brought by RFIM. However, *VMExit* is very frequent (close to 100,000 times per second) during full-virtualization VM execution, and the performance effect can be tolerable via the experiments discussed in Fig. 3.

Algorithm 2 shows the execution procedure of RFIM. f_{syscall} represents a file which the current system call operates on. When the administrator plans to monitor the integrity of certain VM, s_{sig} and s_{sen} are decided by file classification algorithm in Section 3.2. The RFIM tool in user mode will invoke *libxc* library to reset an inexistent value (*inexist_addr*) into register *SYSENTER_EIP_MSR*, and the original value (*saved_addr*) is stored by the VMM. When *VMExit* happens, and if it is caused by page fault, the system call sensor module checks the address. If page-fault linear address (*pfl_addr*) equals to the inexistent

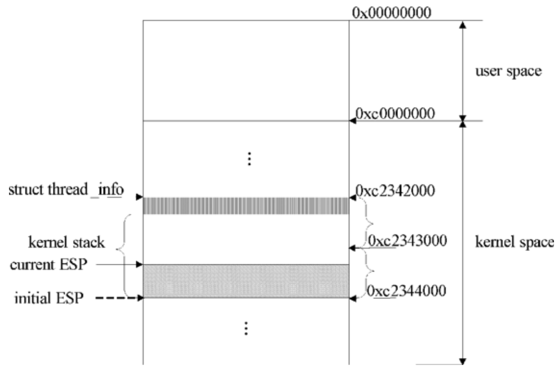


Fig. 4. The thread_info structure instance.

value (*inexist_addr*), it represents that a system call is invoked. The system calls about file modifications are selected, and their parameters are parsed (file name, process identification etc.). s_{syscall} defines all system calls about file modifications in *Linux* 2.6.18.8 (*write*, *open*, *create*, *link*, *unlink*, *mknod*, *chmod*, *utime*, *rename*, *mkdir*, *rmdir*, *truncate*, *ftruncate*, *writev*, *chown*). If the file belongs to s_{sig}, the operation fails. If the file belongs to s_{sen}, this operation is recorded in the database. Otherwise, there is nothing to do. After the VM resumes, the original entry point of the system call handler is assigned to *EIP*. After *VMEntry* happens, the system call handler executes in the VM.

```
Algorithm 2: RFIM algorithm
Input: domain_id, inexist_addr, $s_{sig}$, $s_{sen}$
Output: none

procedure RealtimeMonitoring
    if (domain domain_id doesn't exist)
        exit
    saved_addr = SYSENTER_EIP_MSR
    SYSENTER_EIP_MSR = inexist_addr
    if (VMExit happens)
        if (it is caused by page fault)
            if (pfl_addr = inexist_addr)
            {
                get the system call number (syscall_num)
                if (syscall_num $\in s_{syscall}$)
                {
                    parse the parameters ($f_{syscall}$)
                    if ($f_{syscall} \in s_{sig}$)
                        the operation fails
                    else if ($f_{syscall} \in s_{sen}$)
                        log this operation in the database
                }
                VMWRITE (VMCS.EIP, saved_addr)
            }
end procedure
```

4.3. Semantic reconstruction

When a system call is invoked by an executive program, page fault happens and CPU traps into *root* mode. In the VMM level, *Xen* can access all pages of guest system, but the information in the pages has no semantic to *Xen*. Therefore, it is necessary to reconstruct the semantic of the information expressed in low level.

If page fault is caused by system call according to Algorithm 2, the control flow turns to *Xen*. At the same time, the registers and context of the MVM are saved in *VMCS*. When system calls happen in the MVM, register *EAX* stores the system call number, and the corresponding arguments are held in other registers (*EBX*, *ECX*, *EDX*, *ESI*, *EDI*). There are some system calls with file name, such as *open*. The full path of file manipulated by system call consists of an argument and current directory. However, there are existing system calls with file descriptor as the arguments, such as *read*, *write*. Getting the full path according to file descriptor is the most difficult.

Kernel stack and *thread_info* structure share 2 pages in *Linux*. The *thread_info* locates in lower address, while the kernel stack locates in higher address. That is, if the page size is 4 kB, the *thread_info* structure aligns to 8 kB (2^{13}) and the lower 13 bit is set to zero. Fig. 4 gives an example of *thread_info* structure and kernel stack. The current position of kernel stack is denoted by *ESP* register, which can be acquired from *VMCS*. Based on the discussion above, the initial address of *thread_info*

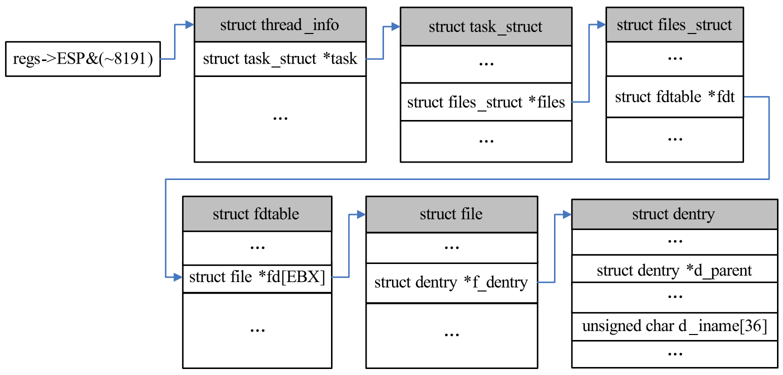


Fig. 5. Get file name from register ESP.

```
struct task_struct {
        ......
        /* process identity */
        pid_t pid;
        ......
        /* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        ......
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                   - access with [gs]et_task_comm (which lock
                                     it with task_lock())
                                   - initialized normally by flush_old_exec */
        ......
        /* open file information */
        struct files_struct *files;
        ......
};
```

Fig. 6. The *task_struct* structure.

structure can be calculated by masking the lower 13 bits of register *ESP*, *ESP* & $(\sim(2^{13}-1))$. By this way, the pointer of *thread_info* structure is reconstructed.

Fig. 5 shows the searching process from register *ESP*. Taking *write* system call, *write*(*fd*, "*test*", *sizeof*("*test*")), as an example: When one application in the MVM launches this system call, it trap into *Xen*, and the state of this VM is saved in the *VMCS* region. The system call number of *write* is 4, which is saved in register *EAX*, and the related arguments are stored in *EBX*, *ECX*, *EDX* in turn. We get the start address of structure *thread_info* via register *ESP*, and one member of the structure points to structure *task_struct*. Structure *task_struct* contains the process state information which is described in Fig. 6, such as process identity (*pid*), process name (*comm*), user identity (*uid*), open file information (*files_struct*). Structure *files_struct* includes the files information used by the current process, and structure *fdtable* is one of its member. Structure *fdtable* contains the current file descriptor table. Due to *fd* in register *EBX*, structure *file* describing current file, contains structure *dentry*, representing file entry. From structure *dentry*, we can get the short path name, and the full path name can be gotten with its member *d_parent* by recursion.

5. Experiments

In this section, we present the experiments to validate the efficiency of our method. RFIM in *VMFence* is implemented on *Xen* 3.2.0. In order to deploy a full-virtualization guest VM, the hardware, especially CPU, must support hardware virtualization extension. The physical platform is two *Pentium* Quad-Core processors, 4 GB of memory, and 160 GB of disk capacity. *Fedora* 8 is preinstalled on the hardware platform, and *Xen* 3.2.0 with modification is compiled and installed on it. We chose *Ubuntu* 8.04 as the guest OS in the full-virtualization VM, that is the MVM. The configuration of the MVM is one virtual CPU and 256 MB memory.

Our system assumes that the VMM layer can isolate the privileged VM from the monitored VM, which is the essential function of virtualization software. This is necessary and reasonable requirement for virtualization-based security technology. Other two preconditions are hardware-based virtualization and fast system call.

(1) Hardware-based virtualization: In order to support full-virtualization VM, there are two different methods: binary translation and hardware virtualization. Binary translation captures these privilege instructions and simulates them in the VMM layer, such as *QEMU*, *VMware*. Hardware virtualization implements the same function via hardware, which

Table 1
The comparison of *LARES*, *XenFIT* and RFIM method.

Aspect	*LARES*	*XenFIT*	RFIM method
Sensor location	The monitored system	The monitored system	VMM
Transparence	No	No	Yes
Isolation	General	Poor	Excellent
Real time	Yes	Yes	Yes

Table 2
The performance of file compression and decompression.

Operation	Native time (s)	RFIM time (s)	Overhead (%)
Compression	203	229	12.8
Decompression	56	62	10.7

Table 3
The performance of encryption and decryption.

	Native time (s)	RFIM time (s)	Overhead (%)
Encryption	58	69	18.9
Decryption	31	37	19.4

has better performance. Hardware-based virtualization is supported by the familiar multi-core CPU, and it is the notable characteristic of current CPU.

(2) Fast system call: Traditional system call is implemented by software interrupt. The switch between user mode and kernel mode consumes hundreds of cycles, and it happens frequently during system execution. Therefore, fall system call has obvious performance advantage compared with traditional system call. Intel x86 CPU supported fast system call instructions (*SYSENTER*/*SYSEXIT*) since PII 300. At present, all mainstream OSes for x86 platform support system call. *Windows XP* adopts fast system call, but *Linux* supports both traditional system call and fast system call at the same time. If guest OS is *Windows XP*, this method will be effective all the time. If fast system call is masked by malicious attackers in *Linux*, the system applies traditional system call by default.

5.1. *Effectiveness*

We compare three systems with similar functions, including *LARES*, *XenFIT* and RFIM method. Table 1 shows the comparison results on four aspects: sensor location, transparence, isolation, and real time. Both *LARES* and *XenFIT* deploy sensors in the monitored system, and intercept system calls. Although *LARES* protects these pages containing hook functions, it loses the transparency to the guest OS. Our method deploys the monitoring point in the VMM, and is full transparent to the monitored system.

Fig. 7 illustrates the control interface of our system running in the privileged VM. It provides the configuration interface for the administrator, and outputs the monitoring results, such as operation, file, user identity, process identity. In order to verify the effectiveness of our prototype in *VMFence*, the rootkit is installed after the monitor is booted. The two file sets (s_{sig} and s_{sen}) have been determined in Section 3.2, and */sbin/init* belongs to s_{sen}. *Mood-NT* [33] is a kernel rootkit on *Linux*. It intercepts the system call table, and replaces */sbin/init* to hide files, processes and network connections. During the system execution, *Mood-NT* is installed to test our prototype.

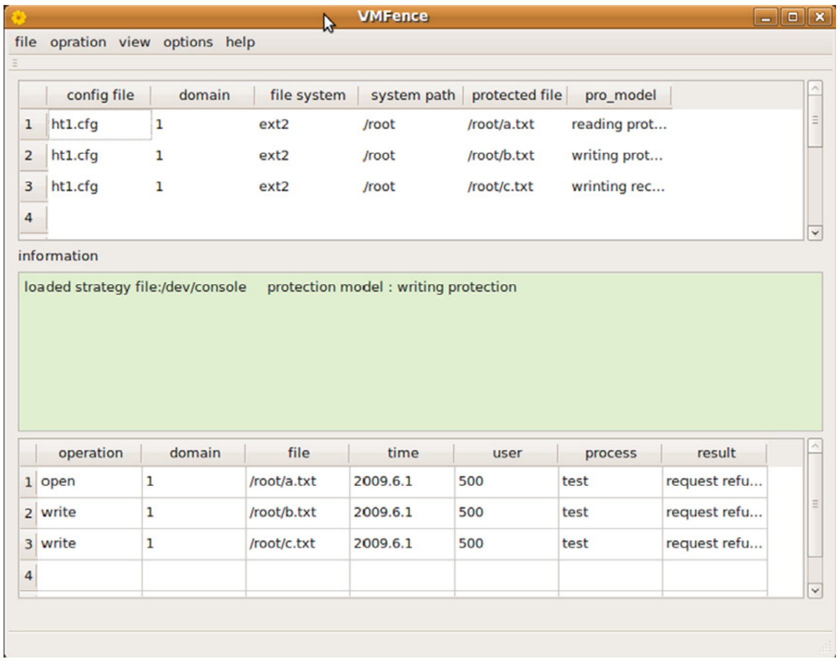
Fig. 8 shows the detailed information when sensitive files are changed in the MVM. This report includes the modification time (Time), the file name (File), the system call (Syscall), process identification (PID), user identification (UID), group identification (GID). *T0rn* [34] is a *Linux* user mode rootkit. It can modify the applications which observe the system state, such as *ls*, *ps*, *netstat*. In our implementation, these applications have been appended to s_{sig}, and this type of attacks can be prevented.

5.2. *Performance analysis*

In this section, the system performance is evaluated on full-virtualization *Ubuntu* 8.04. As file compression and decompression are two common functions for file system, we measure the efficiency of file compression and decompression using *Linux* kernel source. The size of *linux-2.6.18.8.tar.gz* is about 50 MB. We test it on two conditions: one without RFIM (Native), and the other with RFIM (RFIM). The time (seconds) in these two conditions are demonstrated in Table 2. The compression time is far larger than the decompression time. The overhead brought by the RFIM method is less than 13%.

Afterward, we test the performance of file encryption and decryption and *Linux* kernel source (*linux-2.6.18.8.tar.gz*) is taken as test case. Table 3 shows the results of our experiment. The overhead brought by the RFIM method is less than 20%.

iozone is a professional benchmark for file system, which is widely accepted and applied. The command we use is *iozone -a -i 0 -i 1 -n 8K -g 128m*. It indicates that only the efficiency of file reading and writing will be tested. The minimal file



	config file	domain	file system	system path	protected file	pro_model
1	ht1.cfg	1	ext2	/root	/root/a.txt	reading prot...
2	ht1.cfg	1	ext2	/root	/root/b.txt	writing prot...
3	ht1.cfg	1	ext2	/root	/root/c.txt	wrinting rec...
4						

```
loaded strategy file:/dev/console     protection model : writing protection
```

	operation	domain	file	time	user	process	result
1	open	1	/root/a.txt	2009.6.1	500	test	request refu...
2	write	1	/root/b.txt	2009.6.1	500	test	request refu...
3	write	1	/root/c.txt	2009.6.1	500	test	request refu...
4							

Fig. 7. Console of RFIM system.

```
Time:       2009-06-01 21:09:31
File:       /sbin/init
Syscall:    sys_unlink
PID:        16763
UID:        0
GID:        0
```

Fig. 8. Report when sensitive files are modified.

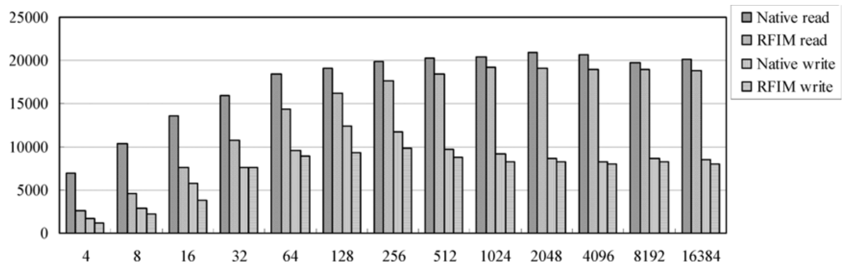


Fig. 9. The performance of *iozone*.

size is 8 kB, and the largest file size is 128 MB. The file size we chose is 16 MB. We focus on both file reading and writing operations, and Fig. 9 shows the speed of file reading and writing when the block size varies from 4 kB to 16,384 kB. The horizontal axis represents the block size (kB), and the vertical axis represents the reading or writing speed (kB per second). The writing speed is almost half of the reading speed. When the block size increases from 4 kB to 16,384 kB, the overall efficiency improves gradually. According to all conditions from Fig. 9, we can know that the reading and writing detection overhead under the RFIM method is a little heavier than the one under native. When the file block size is larger than 128 kB, the performance overhead brought by the RFIM method is less than 15%.

6. Conclusions and future work

In this paper, we have presented a RFIM approach in *VMFence*, which has the merits of strong isolation, transparence, and real time. It does not modify the MVM, and brings acceptable overhead. The files in the MVM are classified into three types: *significant*, *sensitive*, and *ordinary*. The sets are determined by our proposed file classification algorithm. In this paper, the read-forbidden policy is not applied in our design, and the reason is that encryption can achieve this goal better. In *Linux*, the user's home directory and *shell* environment are set according to */etc/passwd* when the user tries to login the system.

The user's password is stored by encryption in */etc/shadow*. The most privileged user, *root* in *Linux*, can change file attributes arbitrarily, so that it is meaningless to set read-forbidden policy for *root*.

Xenaccess can implement FIM without modifying the MVM, but it cannot fetch the detailed information in real time, especially file writing operation. The reason is that file writing operation is optimized by OS in the MVM. We will improve our method for better performance. The next step of *VMFence* is to build an Intrusion Tolerance System (ITS) in virtual computing environment. We focus on the security challenges brought by virtualization technology. Until now, we have implemented prevention from the attacks on the network and file system. In our future work, we will aim at malware detection, especially attacks to VMM, such as VM escape and subverting VMM.

Acknowledgement

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

References

[1] G.H. Kim, E.H. Spafford, The design and implementation of tripwire: A file system integrity checker, in: 2nd ACM Conference on Computer and Communications Security, ACM, Fairfax, 1994, pp. 18–29.
[2] N.A. Quynh, Y. Takefuji, 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, 2007, pp. 194–203.
[3] P. Barham, B. Dragovic, K. Fraser, S.H.T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: 19th ACM Symposium on Operating Systems Principles, ACM, New York, 2003, pp. 164–177.
[4] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, J.N. Matthews, Xen and the art of repeated research, in: 2004 USENIX Annual Technical Conference, USENIX, Boston, 2004, pp. 135–144.
[5] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, A. Mallick, Xen 3.0 and the art of virtualization, in: 2005 Linux Symposium, USENIX, Ottawa, 2005, pp. 65–85.
[6] IDC Virtualization Report, http://www.idc.com/getdoc.jsp?containerId=prUS21473108.
[7] K. Adams, O. Agesen, 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, 2006, pp. 2–13.
[8] J.E. Smith, R. Nair, The architecture of virtual machinese, IEEE Computer 38 (2005) 32–38.
[9] M. Rosenblum, T. Garfinkel, Virtual machine monitors: Current technology and future trends, IEEE Computer 38 (2005) 39–47.
[10] H. Jin, G. Xiang, F. Zhao, D. Zou, M. Li, L. Shi, VMFence: A customized intrusion prevention system in distributed virtual computing environment, in: 3rd International Conference on Ubiquitous Information Management and Communication, ACM, Suwon, 2009.
[11] P.D. Boer, M. Pels, Host-based intrusion detection systems, Technical Report, Informatics Institute, University of Amsterdam, 2005.
[12] A.G. Pennington, J.D. Strunk, J.L. Griffin, C.A.N. Soules, G.R. Goodson, G.R. Ganger, Storage-based intrusion detection: Watching storage activity for suspicious behavior, in: 12th USENIX Security Symposium, USENIX, Washington, 2003, pp. 1–15.
[13] S. Patil, A. Kashyap, G. Sivathanu, E. Zadok, I^3FS: An in-kernel integrity checker and intrusion detection file system, in: 18th USENIX Large Installation System Administration Conference, USENIX, Atlanta, 2004, pp. 67–78.
[14] T. Garfinkel, M. Rosenblum, 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, 2005, pp. 20–25.
[15] R. Perez, R. Sailer, L.V. Doorn, Virtualization and hardware-based security, IEEE Security & Privacy 6 (2008) 24–31.
[16] T. Garfinkel, M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: 10th Network and Distributed System Symposium, IEEE, San Diego, 2003, pp. 191–206.
[17] K. Nance, B. Hay, M. Bishop, Virtual machine introspection observation or interference, IEEE Security & Privacy 6 (2008) 32–37.
[18] S.T. Jones, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, VMM-based hidden process detection and identification using lycosid, in: 4th ACM/USENIX International Conference on Virtual Execution Environments, ACM, Washington, 2008, pp. 91–100.
[19] A. Dinaburg, P. Royal, M. Sharif, W. Lee, Ether: Malware analysis via hardware virtualization extensions, in: 15th ACM Conference on Computer and Communications Security, ACM, Virginia, 2008, pp. 51–62.
[20] R. Riley, X. Jiang, D. Xu, Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing, in: 11th International Symposium on Recent Advances in Intrusion Detection, Springer, Massachusetts, 2008, pp. 1–20.
[21] A. Lanzi, M. Sharif, W. Lee, K-tracer: A system for extracting kernel malware behavior, in: 16th Annual Network and Distributed System Security Symposium, IEEE, San Diego, 2009.
[22] S.T. Jones, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, AntFarm: Tracking processes in a virtual machine environment, in: 2006 USENIX Annual Technical Conference, USENIX, Boston, 2006, pp. 1–14.
[23] X. Jiang, D. Xu, Collapsar: A VM-based architecture for network attack detention center, in: 13th USENIX Security Symposium, USENIX, San Diego, 2004, pp. 1–14.
[24] X. Jiang, X. Wang, D. Xu, Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction, in: 14th ACM Conference on Computer and Communications Security, ACM, Virginia, 2007, pp. 128–138.
[25] X. Jiang, X. Wang, ''Out-of-the-box'' monitoring of VM-based high-interaction honeypots, in: 10th International Symposium on Recent Advances in Intrusion Detection, Springer, Queensland, 2007, pp. 198–218.
[26] B.D. Payne, M. Carbone, M. Sharif, W. Lee, Lares: An architecture for secure active monitoring using virtualization, in: 2008 IEEE Symposium on Security and Privacy, IEEE, California, 2008, pp. 233–247.
[27] B.D. Payne, M.D.P.A. Carbone, W. Lee, Secure and flexible monitoring of virtual machines, in: 23rd Annual Computer Security Applications Conference, IEEE, Korea, 2007, pp. 385–397.
[28] F. Bellard, QEMU, a fast and portable dynamic translator, in: 2005 USENIX Annual Technical Conference, USENIX, Anaheim, 2005, pp. 41–46.
[29] G. Neiger, A. Santoni, F. Leung, D. Rodgers, R. Uhlig, Intel virtualization technology: Hardware support for efficient processor virtualization, Intel Technology Journal 10 (2006) 167–177.
[30] Intel staff, Intel 64 and IA-32 Architectures Software Developer's Manuals, Intel Corporation, November 2008.
[31] AMD staff, AMD64 Architecture Programmer's Manual, AMD Corporation, September 2007.
[32] D.P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd ed., O'Reilly, Sebastopol, 2005.
[33] Mood-NT, http://darkangel.antifork.org/codes.htm.
[34] Analysis of the T0rn Rootkit, http://www.securityfocus.com/infocus/1230.