CFWatcher: A Novel Target-based Real-time Approach to Monitor Critical Files using VMI

Dongyang Zhan*, Lin Ye*, Binxing Fang*, Xiaojiang Du† and Shen Su*
*Harbin Institute of Technology
Email: {dongyangzhan, yelin, bxfang, sushen}@pact518.hit.edu.cn
†Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA
Email:dxj@ieee.org

***Abstract*—Protecting critical files in file systems is very important to computer systems. To protect critical files, the VMI-based Real-time File-system Monitor tools are promising options. However, these tools are always operation-based and introduce high overhead. The operation-based approaches intercept some kind of file operation to monitor critical files. The selected file operation is intercepted by the monitor whenever it is being executed. As file operation are high-frequency, the operation-based methods always result in the high performance degradation. In this paper, we present a VMI-based low overhead real-time critical file monitor method, CFWatcher, to meet the performance requirements of real-time monitor tools. CFWatcher is a target-based monitor tool which means it only intercepts the file operations accessing the user-defined critical files, and then obtains enough information to check the rules. The overhead of CFWatcher is related to the frequency of the target being accessed. Besides monitoring critical files, CFWatcher can take actions to prevent the illegal access if there is any rule violation. We implemented the prototype of CFWatcher and then evaluated the performance. Experimental results show that the overhead of our approach is low.**

***Keywords*—Critical file monitor, target-based method, VMI.**

I. Introduction

Protecting critical files in file systems is very important to computer systems. Most of the attacks work through unauthorized access to critical files to steal the confidential information like password, credit-card number etc. and then they usually hide their traces by subverting critical files, such as system logs. File-system Monitor is a popular approach to protect critical files. Traditional File-system Monitor [1]–[4] is usually an agent or a kernel module running in the operating system. As the malware are also running in the operating system or even in the kernel, these in-the-box approaches take risks of the monitor being detected and the monitor being subverted by the malware.

Virtual Machine Introspection(VMI) gives people a novel way to reduce these risks. Under the VMI architecture, monitor applications are always deployed in VMM or another protected VM. The monitoring sensors are moved to the VMM, so they can't be detected by the target VM. VMM protects the monitor applications from being subverted. VMI-based File-system Monitor tools can be classified into two types: periodic monitor and real-time monitor. Periodic monitor tools [5]–[7] compare current attributes of the critical files with previously gathered, such as the owner, the content, and the last modification time etc. These approaches can't protect critical files in real time. The attacker could modify and then recover the critical files during the sleep period. Real-time monitor tools [8]–[11] intercept file operations during the execution process of the system. They always hook the system calls or the backend driver related with file operation to get attributes of the operated file, such as file name, and then compare them with the blacklist. These methods are operation-based, because they first intercept every file operation that can access the monitored files, and then check whether the operated file should be monitored. File operation is high-frequency in operating systems, but the operation accessing target files is always low-frequency, so the operation-based methods' overhead is high, and most of the overhead is meaningless.

In this paper, we present a VMI-based target-based real-time critical file monitor method, CFWatcher, to meet the performance requirements of real-time monitor tools. CFWatcher is transparent and secure as it is a VMI-based solution. All the sensors run in the VMM layer, so CFWatcher is isolated from the target VM and transparent to the target VM. As virtualization technology is widely used by cloud computing, CFWatcher could also be used in a cloud environment to protect critical files in VMs.

CFWatcher is a low-overhead monitor tool, because it is a target-based approach. The target-based approach means the monitor tool's interceptor works only when the target is operated. The overhead of a target-based monitor tool is related to the frequency of the target being accessed, instead of the frequency of the target VM's operation. As a target-based tool, CFWatcher only intercepts the file operation accessing the user-defined critical files, and then obtains enough information to check the rules, such as process id, file name.

Besides monitoring predefined critical files, CFWatcher is able to take actions to prevent the illegal access when there is any rule violation. CFWatcher's administrators can define many kinds of rules, because the interceptor can get comprehensive information of the target process. To prevent the illegal access, CFWatcher makes the illegal file operation fail, so that the attackers can't operate the target files any more.

We implemented the prototype of CFWatcher on XEN with full virtualization using the Volatility Framework [12]. In our implementation, only Ubuntu was used as the target VM, as it is a widely used operating system, but CFWatcher is an adaptable monitor tool which can monitor most of Linux VMs

978-1-4799-6664-6/16/$31.00 ©2016 IEEE

without any modification.

The rest of this paper is organized as follows: Section 2 covers the related work. Section 3 introduces the design of CFWatcher. The implementation of CFWatcher is described in Section 4. Section 5 evaluates the performance of CFWatcher. Conclusions and future work are presented in Section 6.

II. Related Works

As file-system integrity is important to the overall security, it is not surprising that there are many papers and resultant tools on the topic. In this section, we introduce these works respectively.

Traditional File-system Monitor is usually an agent or a kernel module running in the operating system. Tripwire [1] is an integrity checking tool designed for the UNIX environment to let system administrators monitor their file systems for unauthorized modifications. It creates a database to store some unique identifier for each file to be monitored. It is possible to determine if a file has been modified by comparing it with the saved version. Furthermore, it is possible to determine if files have been added or deleted from the system. Unlike Tripwire, I^3FS [2] runs in OS's kernel intercepts system calls and compares the checksums of files in real-time. XenRIM [3] is deployed in XEN environment, the agents are running in VMs to intercept the file operation, and the server is running in DOM0 to receive logs sent by agents. Flogger [4] can be implemented in both VM and PM kernels. It intercepts file operations and then writes events into log files. As the malware are also running in the operating system or even in the kernel, these in-the-box approaches take risks of the monitor being detected and the monitor being subverted by the malware.

As virtualization technology is widely applied to various aspects of computer systems and cloud environment. VM-based security [13], [14] becomes increasingly important. Virtual Machine Introspection [15] (VMI) is a powerful technique that allows monitor a running VM's execution without any agents. VMI is widely used to protect virtual machines [16]–[18].

VMI-based File-system Monitor tools can be classified into two types: periodic monitor and real-time monitor. Periodic monitor tools always compare current attributes of the critical files with previously gathered attributes, such as the owner, the content, and the last modification time etc. They also compare files on the disk with the black list to find malwares. VMWatcher [5] exports the files in the target VM to the trusted VM (DOM0), and then carries out malware detection through anti-virus tools deployed on the host. CFMT [7] calculates cryptographic checksum of each file and stores it in file itself, and check each file's encrypted hash periodically. VMDriver [6] periodically checks file objects that the current process operates on, and lists all operations on specific files, such as process id, process name, system call number, file name, and operation time. The periodic method's problem is it cannot detect the modification in time and the attacker could modify and then recover the critical files during the monitor tool's sleep period.

Real-time VMI-based monitor tools are more popular than periodic tools, because they can protect critical files in real-time. Real-time monitor tools intercept file operations during the execution process of the system. XenFIT [9] is a file integrity monitor designed for XEN VMs. Breakpoints are inserted in the monitored system to intercept system calls related with file operations, for instance, open, close, etc. Unlike XenFIT, [10] proposed a guest-transparent RFIM, which intercepts file operation in VMM. It could get the file operation information, like process identity, file name, file operation, time et al., when one critical file is modified in the target VM. [8] deploys sensors in the target VM's VFS file operation functions to capture the operated file name. [11] is a XEN-based secure virtual disk access-control method, it works in the Qemu-dm which exists in DOM0, and corresponds with every virtual system's device daemon. This method lets Qemu-dm only handle secure I/O requests. These methods are all operation-based. When some kind of file operation is used to monitor critical files, it is intercepted by the monitor tool whenever it is being executed. File operation is high-frequency in operating systems, but operations accessing target files are always low-frequency, so the operation-based method's overhead is high, and most of the overhead is meaningless. Our paper aims to design and implement a target-based user-defined critical file monitor approach using VMI technology.

III. Design

CFWatcher is a VMI-based monitor tool working out of the box. It is used to monitor file operations on the predefind critical files in guest VM. The list of critical files and the rule corresponding to these files can be defined by the administrator.

A. Architecture

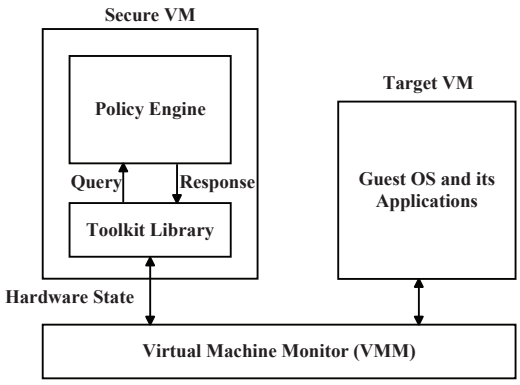

Fig. 1. A High-Level View of CFWatcher Architecture.

As shown in Figure 1, on the right is the monitored virtual machine (target VM). On the left is the VMI-based CFWatcher with its major components: the toolkit library that provides an OS-level view of the VM for the policy engine by interpreting the hardware state exported by the VMM, the policy engine consisting of all the policies. The virtual machine monitor (VMM) isolates the monitor tool from the VM to be monitored and allows the monitor tool to inspect the hardware state of the target VM. The VMM also allows the monitor to change the content of the VM's memory and registers.

Policy engine is the core of our system. There are three functions in CFWatcher's policy engine: 1) monitoring critical

files, 2) creating dentry and inode objects, 3) checking rules and taking actions in response to rule violations. We will describe the detailed design of these functions in the following paragraphs.

B. Target-based Monitor Mechanism

CFWatcher monitors critical files by monitoring the operation of dentry and inode objects corresponding to critical files.

In Linux, all files are accessed through the Virtual File System [19] (VFS). The concept of a directory entry (dentry object) is employed by VFS. Dentry objects are all components in a path, including files, they make the software accessing files easier and faster. When userspace software accesses a file, VFS uses the file's pathname to search through the directory entry cache (also known as the dentry cache). If the corresponding dentry object has not been created, VFS will create it. After each element in the filepath is resolved into a dentry object by VFS and arrives at the end of the filepath, the kernel caches these dentry objects in the dentry cache. By searching the dentry cache, it is very fast to translate a file name into a specific dentry object. Dentry objects are never saved to disk, they only live in memory for performance.

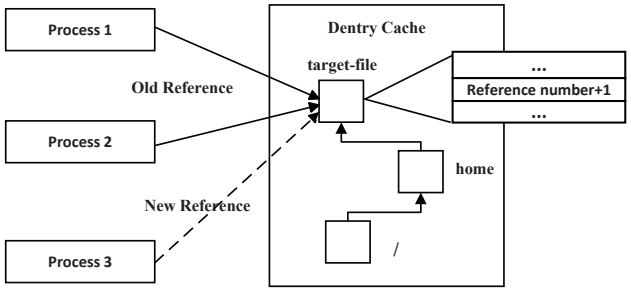

Fig. 2. Dentry cache.

A used dentry object points to valid data and can be used by one or more users, so than it cannot be discarded. A dentry object can be freed, when it is not currently used by VFS. In that case, there is no software accessing the file corresponding to the dentry object. In order to count how many users are using the dentry object, every dentry object has a field to record the number of valid references. As shown in Figure 2, when a new reference comes, the dentry object's reference number increases. Although there are many modes which can be used by software to open files, there must be a new reference to the corresponding dentry object.

CFWatcher monitors critical file access activities by monitoring the corresponding dentry object operation. First, CFWatcher finds the region of the monitored files' corresponding dentry objects in VM's memory (RAM) by using the target VM's OS-level semantic information. Second, CFWatcher finds and records the fields of every dentry object's reference number in these memory regions. At last, CFWatcher monitors the changes of these memory fields. If one of the monitored field's value increases, which means someone is accessing the corresponding file, CFWatcher can intercept the event immediately. If the number of reference is decreased to zero, the monitored dentry object may be discarded. To make the monitored dentry objects living in RAM forever, CFWatcher adds the original value before it down to zero using VMI technology. In that case, although there is no valid reference to the dentry object, the reference number of it is still one, and it can't be freed.

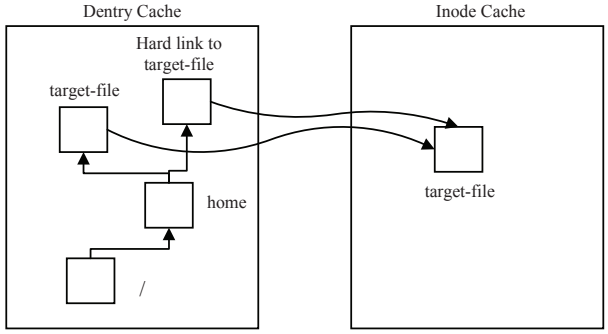

Fig. 3. Hard link.

An individual dentry object is usually pointed to an inode object. Dentry objects only live in RAM, but inode objects live on the disk. Every file on the disk is corresponded to only one inode object. When a file is accessed, the corresponding inode object is copied into the memory, and then changes to the inode are written back to the disk. Hard link makes a single inode object be pointed to by multiple dentry objects as shown in Figure 3. Every inode object records the number of hard links to itself. When the number is greater than one, there may be many dentry objects pointing to the object. These dentry objects are linked by a doubly linked list. To ensure nobody can bypass our monitor system using hard link, CFWatcher finds all the dentry objects to the target file by following the linked list and monitors them.

When a file is being removed, the related inode object's link number is decreased to zero. CFWatcher can intercept the deletion of the target files by monitoring the corresponding inode object's link number. The method is similar to monitoring the dentry object's reference number. When one of the monitored field's value decreases, CFWatcher intercepts the removing of the corresponding file immediately.

Attachers may bypass CFWatcher by directly operating the storage devices, but the storage devices are still files in VFS (for example, in Ubuntu, the first hard-disk's file descriptor is like "/dev/xvda"). So CFWatcher can monitor the devices' objects to protect these attacks. Kernel attack is another kind of method to subvert CFWatcher, these attacks may change the kernel or even replace the kernel. As CFWatcher is not a full-featured IDS, it can't prevent these attacks, but we can defeat these attacks by using other kernel protection tools.

C. Creating Dentry and Inode Objects

CFWatcher monitors the operation of the target files by monitoring the corresponding dentry objects and inode objects. If the target dentry objects and inode objects are not in cache, CFWatcher can't perform functions. We can't guarantee that the target files have been accessed before CFWatcher starts, so CFWatcher should be able to create the corresponding dentry objects and inode objects which are not in the VM's memory.

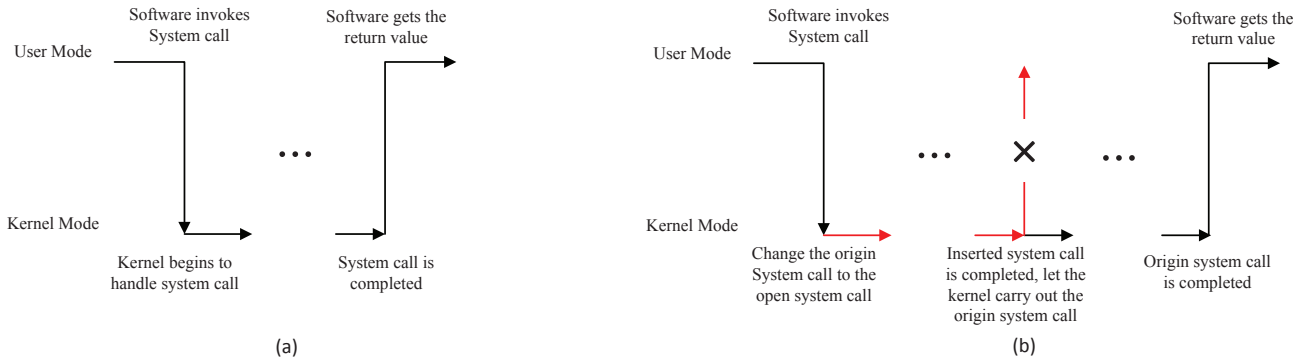

Fig. 4. Normal and inserted system call flow.

When a file that has never been accessed before is being opened, the VFS creates the corresponding dentry object and inode object in the VM's memory. To ensure the existence of monitored files' dentry objects and inode objects in memory, using an agent running in target VM to open all the monitored files is an easy solution, but an agent-based approach is not safe, because the agent could be detected and be subverted.

In our solution, we leverage an out-of-box approach to create dentry and inode objects of the monitored files in the target VM's memory. In a modern operating system, all of the important resources (file, network etc.) can only be accessed via system call, so the OS handles system calls with very high frequency. The core idea of our approach is to let the target VM's OS carry out an extra open system call. This extra system call opens the file that should be monitored. When the OS wants to handle a system call invoked by the userspace software, it enters kernel mode then handles the system call as shown Figure 4 (a). To achieve our goal, we select the moment when the target VM's OS has just entered kernel mode to insert our extra system call.

As shown in Figure 4 (b), our insert approach has two steps.

Step 1: We change the origin system call to the open system call, which opens the file that should be monitored.

Step 2: When our inserted system call has just completed, we let the target VM's OS carry out the origin system call before it re-enters user mode. In this case, the userspace software gets the right return, so it can't detect the inserted extra system call.

D. Checking Rules & Taking Actions

CFWatcher's rule checker module is triggered by the interceptor when the monitored files are operated. After being triggered, rule checker gets enough information of the process which is accessing the critical files using VMI technology. The process information includes the process name, process id, user id, group id etc. Rules of the rule checker module are defined by CFWatcher's administrators. Administrators can define many kinds of rules. For example, the administrator defined a rule that "/test/test.txt" can only be accessed by root user with the "cat" process in VM. If a userspace process opens the target file, the rule checker module checks whether the process's user id is root and the process's name is "cat". If it is an illegal operation, CFWatcher takes actions to stop it.

To prevent the illegal access, CFWatcher makes the target system call fail by setting the return value of it to a negative number and clearing the related file descriptor in the target process. The userspace process can't access the target file anymore because the open operation failed and the file descriptor doesn't exist. To prevent remove operation on the critical files, CFWatcher should also ensure the file can't be removed on the disk. To that end, CFWatcher adds the link number value before it is going to be decreased to zero. The userspace software can't delete the critical file illegally, because the link number of the corresponding inode object can never be decreased to zero.

IV. Prototype Implementation

A. Implementation Environment

The current CFWatcher prototype is implemented on an x86 server that supports Intel VT [20] technology and is designed to support guest VMs running all contributions of Linux OS. An open-source hypervisor, XEN [21], is selected as the VMM. Ubuntu is selected as the guest VM.

We leverage the Volatility Framework [12] to provide an OS-level view of the guest VM. CFWatcher is an adaptable monitor tool which can monitor most of Linux VM without any modification, because the Volatility Framework is a programmed OS-level semantic analysis toolkit. The kernel symbols and data structure are contained in its built-in profile, so CFWatcher need not manually analyze the binary layout of the target VM.

B. Monitor Initialization

CFWatcher first uses XEN toolkit library to access the memory of the monitored VM. This library exposes the memory of a VM to the Dom0, so we can read the live VM's memory in the Dom0. This memory address space (AS) can be used by the Volatility Framework. We extended the Volatility framework with a new plugin called dentryfinder to find the target dentry objects in dentry cache. The dentryfinder plugin is written in Python and utilizes the existing Volatility scanning algorithms for extraction of kernel data structures from the target VM's memory. The dentryfinder plugin analyzes the

target VM's memory by reading the AS which is mapped by XEN toolkit library. It calculates offsets and lengths of data fields of dentry objects using Volatility's built-in profiles. The dentryfinder plugin is able to identify all invalid dentry structures within target VM's dentry cache. If the dentryfinder finds the target dentry objects, it reports the address of them.

C. Monitoring Special Memory Region

CFWatcher monitors the modification and execution of special memory region by setting the Extended Page Table (EPT) entry of the target VM's physical page which contains the monitored region to read-only (to intercept the writting events) or read-only and write-only (to intercept the executing events) [22]. When the target page is accessed, an EPT violation is triggered which can be captured by CFWatcher. If the accessed address belongs to the region to be monitored, CFWatcher can invoke the predefined monitor handler. Then, CFWatcher sets the corresponding EPT entry to writable and resumes the target VM with single-step mode, so that the target VM can access the monitored memory region. After the target VM executes the memory access instruction, CFWatcher resets the corresponding EPT entry to read-only or write-only and read-only to intercept future access.

D. Creating Dentry and Inode Objects

If there is no dentry object corresponding to the target file, CFWatcher creates it. As mentioned in section 3, CFWatcher inserts an open system call to create these objects. To that end, we exploit the features of the Intel fast system call entry mechanism.

In Intel x86 architecture, SYSENTER/SYSEXIT instructions [23] are used for fast entry to the kernel. SYSEXIT is a companion instruction to SYSENTER. The SYSENTER instruction is used by system calls to convert user mode to kernel mode. When SYSENTER is executed, the CPU switches to ring 0, and begins to execute the system call procedure. When SYSEXIT is executed, the CPU re-enters ring 3. The CPU doesn't save state information for the user code when executing a SYSENTER instruction.

We change the system call number and parameters by changing CPU registers. Nether the SYSENTER nor the SYSEXIT instruction uses the stack to pass parameters. Instead, SYSENTER and SYSEXIT leverage the CPU registers to pass parameters. In SYSENTER, EAX stores the system call number, EBX stores the first argument, and ECX stores the second argument, and so on. There can be six arguments, max. SYSEXIT leverages EAX to pass the system call return value. It is easy to change the system call number and parameters by changing registers when SYSENTER is executing, and system call return value when SYSEXIT is executing.

V. Performance Evaluation

We evaluated the performance of our system in this section. Our testbed consisted of a virtualized server, whose hypervisor was XEN version 4.3 and Dom0's OS kernel was Ubuntu 13.10. The host system had Core i5 processor running two cores at 2.4GHz and 4GB of system memory. The CFWatcher system was installed in the Dom0 domain. In addition, the virtualized server hosted one full-virtualization VM running a

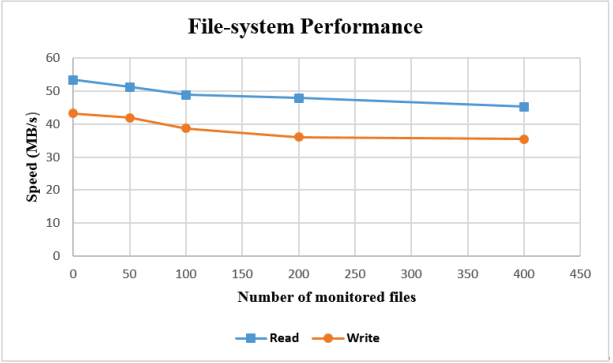

Fig. 5. File-system Performance.

default installation of Ubuntu 12.04. The VM was configured with 512MByte RAM and 1 virtual CPU (VCPU).

We analyzed the overhead introduced by CFWatcher on file-system. To protect the user's confidential information, we only need to monitor a small number of critical files in the target VM. However, we are interested in the performance overhead as the number of monitored files increases. As our test VM totally opened about 900 different files, so we evaluated the performance impact for up to monitoring 400 different files simultaneously, this is about half of the opened files.

We used the Linux dd command to assess the file-system performance. dd is a useful and simple file-system benchmark tool, which is widely accepted and applied. dd is a command-line utility for Unix and Unix-like operating systems whose primary purpose is to convert and copy files. In Linux, device drivers for hardware (such as hard disks) and special device files (such as /dev/zero and /dev/null) appear in the file system just like normal files, dd can also read and/or write from/to these files. /dev/zero can provide as many null characters (ASCII NUL, 0x00) as are read from it, so copying data from it only has write overhead. /dev/null is a device file that discards all data written to it but reports that the write operation succeeded, copying data to it only produces read payload. As a result, dd can be used to generate and measure the read and write operations.

In our test, write and read operations were measured. We tested read speed by using the command "dd -bs 128k count=10240 if =/dev/xvda of=/dev/null", which means copy 10k blocks from the disk to /dev/null, the size of each block is 128kByte. Then we used "dd -bs 128k count=10240 if =/dev/zero of=testfile" to test write speed. In this case, dd copid 1.3GByte data from /dev/null to the disk. We set the total size of copied data to 1.3Gbyte which is twice the size of the target VM's RAM to reduce errors generated by it.

We first executed the test process without CFWatcher, then ran it again with CFWatcher. We did each test 10 times and recorded the average value. Figure 5 shows the read and write performance of the VM without CFWatcher and under CFWatcher with 50 to 400 monitored files. The horizontal axis represents the number of files to be monitored, and the vertical axis represents the reading or writing speed (MB per second).

According to all conditions from Figure 5, we can know

when the number of monitored file is small, the overhead of monitoring is low. For the 50 monitored files in two experiments, the file-system overhead of monitoring as compared to an unmonitored system was less than 5%. And in the case of 100 monitored files, the file-system overhead is less than 9%. In Figure 5, the max read performance decrease is 15% in the case of 400 monitored files. For write, the max performance decrease is 20% in the case of 400 monitored files.

When the number of monitored files increased, the number of monitored memory regions also increased. In XEN, We can only monitor the modification of the target VM's memory page, so that the memory page performance decreases if it contains a monitored region. As a result, large numbers of monitored files result in the decrease of the VM's cache performance.

When the target file is being accessed, the rule checker module is triggered to check rules. We also measured the performance of the rule checker module. To simplify the test, the rule checker module only records the access log. An agent was developed running in the target VM, which opens the monitored files 100 times and records the average processing time. We first executed the agent without CFWatcher, then ran it again with CFWatcher, and finally calculated the difference between the two values. We did this test 10 times, the average overhead of the rule checker module is 2 ms.

VI. Conclusion

In this paper, we have presented the design, implementation and evaluation of CFWatcher, a VMI-based real-time critical file monitor with low-overhead. CFWatcher is a target-based monitor tool which means it only intercepts the file operation accessing the critical files, and then obtains enough information to check rules. We also presented a no-agent approach to create the dentry and inode objects corresponding to the monitored files which ensures CFWatcher can monitor all the critical files. After implementing the prototype of CFWatcher on XEN with full virtualization, we have evaluated the performance of it. From the experiment, the overhead introduced by CFWatcher increases with the number of monitored files. When the number is below 100, the overhead is less than 5%; and in the case of 200 monitored files, the overhead is less than 9%.

In the future work, we are going to extend our system to a full-featured IDS. As Windows is a popular operating system, we are also going to extend our system to protect files in VM running Windows.

Acknowledgement

This work is supported by National Natural Science Foundation of China (Project NO. 61173144).

References

[1] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM, 1994, pp. 18–29.

[2] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok, "I3fs: An in-kernel integrity checker and intrusion detection file system." in *LISA*, vol. 4, 2004, pp. 67–78.

[3] N. A. Quynh and Y. Takefuji, "A real-time integrity monitor for xen virtual machine," in *Networking and Services, 2006. ICNS'06. International conference on*. IEEE, 2006, pp. 90–90.

[4] R. K. Ko, P. Jagadpramana, and B. S. Lee, "Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011, pp. 765–771.

[5] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[6] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao, "Vmdriver: A driver-based monitoring mechanism for virtualization," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 72–81.

[7] S. Gupta, A. Sardana, and P. Kumar, "A light weight centralized file monitoring approach for securing files in cloud environment," in *Internet Technology And Secured Transactions, 2012 International Conference for*. IEEE, 2012, pp. 382–387.

[8] K. Asrigo, L. Litty, and D. Lie, "Using vmm-based sensors to monitor honeypots," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 13–23.

[9] N. A. Quynh and Y. Takefuji, "A novel approach for a file-system integrity monitor tool of xen virtual machine," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 194–202.

[10] H. Jin, G. Xiang, D. Zou, F. Zhao, M. Li, and C. Yu, "A guest-transparent file integrity monitoring method in virtualization environment," *Computers & Mathematics with Applications*, vol. 60, no. 2, pp. 256–266, 2010.

[11] F. Liu, H. Zhang, and H. Zhou, "A xen-based secure virtual disk access-control method," in *2010 International Conference on Multimedia Information Networking and Security. Nanjing, Jiangsu China: IEEE Computer Society*, 2010, pp. 375–378.

[12] "The volatility foundation," http://www.volatilityfoundation.org/.

[13] R. Perez, L. van Doorn, and R. Sailer, "Virtualization and hardware-based security," *IEEE Security & Privacy*, no. 5, pp. 24–31, 2008.

[14] T. Garfinkel and M. Rosenblum, "When virtual is harder than real." HotOS, 2005.

[15] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[17] J. Hizver and T.-c. Chiueh, "Real-time deep virtual machine introspection and its applications," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2014, pp. 3–14.

[18] A. F. Fanton, J. J. Gandee, W. H. Lutton, E. L. Harper, K. E. Godwin, and A. A. Rozga, "Cloud-based application whitelisting," Nov. 29 2011, uS Patent 8,069,487.

[19] "Vfs," http://wiki.osdev.org/VFS.

[20] P. Guide, "Intel® 64 and ia-32 architectures software developers manual," 2010.

[21] "The xen project," http://www.xenproject.org/.

[22] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 289–298.

[23] "Sysenter," http://wiki.osdev.org/SYSENTER.