PAPER Special Section on Security, Privacy and Anonymity in Computation, Communication and Storage Systems

Protecting Critical Files Using Target-Based Virtual Machine Introspection Approach

Dongyang ZHAN[†a)], ***Student Member***, **Lin YE**[†b)], **Binxing FANG**[†c)], **Xiaojiang DU**[††d)], ***and*** **Zhikai XU**[†e)], ***Nonmembers***

SUMMARY Protecting critical files in operating system is very important to system security. With the increasing adoption of Virtual Machine Introspection (VMI), designing VMI-based monitoring tools become a preferential choice with promising features, such as isolation, stealthiness and quick recovery from crash. However, these tools inevitably introduce high overhead due to their operation-based characteristic. Specifically, they need to intercept some file operations to monitor critical files once the operations are executed, regardless of whether the files are critical or not. It is known that file operation is high-frequency, so operation-based methods often result in performance degradation seriously. Thus, in this paper we present CFWatcher, a target-based real-time monitoring solution to protect critical files by leveraging VMI techniques. As a target-based scheme, CFWatcher constraints the monitoring into the operations that are accessing target files defined by users. Consequently, the overhead depends on the frequency of target files being accessed instead of the whole filesystem, which dramatically reduces the overhead. To validate our solution, a prototype system is built on Xen with full virtualization, which not only is able to monitor both Linux and Windows virtual machines, but also can take actions to prevent unauthorized access according to predefined policies. Through extensive evaluations, the experimental results demonstrate that the overhead introduced by CFWatcher is acceptable. Especially, the overhead is very low in the case of a few target files.

key words: *Monitoring, VMI, target-based, filesystem*

1. Introduction

Protecting critical files in modern operating system (OS), such as drivers, configurations, confidentialities et al, is very important to system security. Many attacks exploit unauthorized access to critical files as a prelude for more advanced forms. For example, the file */dev/mem* may be tampered to compromise Linux kernel. As a result, many efforts have been made to build up filesystem monitors. Traditional approaches [1]–[3] usually employ an agent or a kernel module installed in the OS to detect what is happening to critical files. Unfortunately, as malwares are running in the OS (or even in the kernel), these in-the-box approaches take risks of the monitor being detected and then subverted by malwares, which finally circumvents the detection.

As an emerging technique, Virtual Machine Introspection (VMI) offers a promising way to tackle aforementioned dilemma. Within VMI-enabled architecture, monitoring functions can be immigrated from Virtual Machine (VM) to Virtual Machine Monitor (VMM). Since VMM is more privileged and also transparent to VM, VM-level malwares are difficult to be aware of or even attack VMM-level agents. As far as we know, VMI-based filesystem monitoring tools can be classified as: polling scheme [4], [5] and event-driven scheme [6]–[11]. The former examines file integrity in each interval, but leaves a gap between checkpoints exploited by malwares to modify the files transiently and then roll back without being detected, which undermines its effectiveness. In comparison, event-driven scheme usually monitors specific file operations at runtime. When every operation occurs, it will be intercepted to ensure whether it should be permitted. However, as file operation is high-frequency in the OS, such operation-based scheme has to suffer from performance degradation caused by all files rather than critical files that take up a small percentage. Heavy overhead makes operation-based scheme less practical in real world.

To meet performance requirements of monitoring tools, such as real-time and low-overhead features, we devise and implement target-based approaches to monitor critical files in both Linux and Windows by leveraging VMI, namely CFWatcher [12]. The term target-based means that the monitor will be triggered only if accessing the target files (open, deletion et al). The obvious advantage beyond operation-based scheme is that the overhead mainly depends on the frequency of accessing target files rather than that of file operations on all files, which can dramatically reduce the overhead. In order to enable target-based monitoring, we investigate into the internal mechanisms of Linux and Windows filesystems in depth, and locate the relevant operations when accessing the target files. Our core idea is to monitor the operations on meta objects of target files in VM's memory instead of themselves. When the target files are being accessed, the meta objects will also be operated. Afterwards, monitoring these operations in VMM can guarantee that accessing other files will not trigger the monitor.

To enhance CFWatcher's abilities, we take a further step in two aspects. First, we allow users to define the set of critical files on their own and even express complex rules. At runtime CFWatcher can take actions to protect critical files from unauthorized access by determining whether any

Manuscript received December 6, 2016.
Manuscript revised June 8, 2017.
Manuscript publicized July 21, 2017.

[†]The authors are with the School of Computer Science Engineering, Harbin Institute of Technology, Harbin, 150001, China.
[††]The author is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA.
a) E-mail: zhandy@hit.edu.cn
b) E-mail: hityelin@hit.edu.cn
c) E-mail: bxfang@nis.hit.edu.cn
d) E-mail: dxj@ieee.org
e) E-mail: zhikaixu@foxmail.com

DOI: 10.1587/transinf.2016INP0009

predefined rules are violated or not. Second, we harness automatic semantic analysis tools that can release the complexity from binary analysis manually, to render CFWatcher well compatible with most of Linux and Windows distributions without any modification.

The main contributions of our work are as follows:

- We propose CFWatcher, a target-based event-driven approach to monitor critical files in both Linux and Windows, which allows users to define the rules on their own so as to determine whether unauthorized access is permitted in a timely and proper way. CFWatcher also demonstrates several advantageous features, such as transparency to VM and isolation from VM attack vectors.
- CFWatcher prototype is implemented in a hardware-assisted virtual environment within good compatibility via automatic semantic analysis tools that extract meaningful information from binary memory data.
- The effectiveness and performance of CFWatcher is evaluated extensively, which proves that its overhead is acceptable, especially when the number of target files is low.

This paper is extended from [12], which proposed a target-based critical file monitor for Linux VMs. In this paper, we first extend CFWatcher to Windows VMs, which means CFWatcher can also monitor and protect the Windows VM files in a target-based way. Second, we proposed a novel agent-based method to create cache objects supporting both Windows and Linux VMs. Third, we present the detailed design and implementation for Windows VMs. Fourth, the effectiveness and performance of CFWatcher in Windows VMs are also evaluated and discussed. Furthermore, we compare CFWatcher with operation-based and commercial approaches. Finally, we discuss the security analysis and potential countermeasures.

The rest of this paper is structured as follows: Sect. 2 summarizes the related work. The design of CFWatcher is introduced in Sect. 3. Section 4 states the implementation comprehensively. Section 5 evaluates the effectiveness and performance of our solution. Section 6 discusses security analysis and potential countermeasures with conclusions and future work presented in Sect. 7.

2. Related Work

Since file integrity is important to computer security, there have been many efforts devoting to the topic. Traditional filesystem monitor usually acts as an agent or a kernel module running in the OS. For example, ICAR [1] works as an Linux kernel module, and checks for file intrusion. It can recover the original version of the subvirted file from a backup. Unlike ICAR, [2] stores all of the crucial data in a physically write-protected storage, and uses them to check file integrity. XenRIM [3] runs in Xen environment where the agents are running in VMs to intercept file operations and the server is running in Dom0 to receive logs sent by agents. Unfortunately, these in-the-box approaches take risks of the monitor being detected and subverted by the coexisting malwares.

In recent years VMI-based security tools becomes more popular increasingly. As a promising technique, VMI [13] demonstrates its advantages in many areas since it allows administrators to monitor a running VM's execution in an out-the-box way, such as virtual machine monitoring [14], virus analysis [15] and intrusion detection [16].

Generally, VMI-based filesystem monitoring tools can be divided into two types: polling scheme and event-driven scheme. Polling scheme compares current file attributes with those previously gathered periodically, such as the owner and content etc. It also can find malwares on the disk by using the blacklist. [4] works in visualization environment and compares low security level file with the baseline databases. CFMT [5] stores each file's checksum into file itself and periodically checks the integrity of them. In summary, the key to polling scheme is the tradeoff between accuracy and performance in term of inspection cycle. If the cycle is too short, high-frequency inspection will pose a serious impact on system performance. Otherwise, it cannot detect the variation in a large cycle. The attacker could modify and then recover the critical files immediately.

By contrast, event-driven monitoring tools are more popular because they can protect critical files by intercepting file operations during the execution. They often hook system calls or backend drivers to get the attributes of the operated target files. Flogger [6] can be implemented in both VM and PM (physical machine) kernels, which captures file operations and then records the events in log files. [7] and [8] intercept file operations in VMM. The breakpoints are implanted into target system, and hooks system calls related with file operations, such as open and close. vMon [9] hooks QEMU I/O handler to achieve secure check and designs File-to-Block Mapper to bridge the semantic gap between the disk level and Linux file system level. [10] works in the Qemu-dm which exists in Dom0, and analyzes every virtual machine's I/O request to make sure they are secure. For Windows VMs, Filesafe [11] intercepts every read or write request to disk blocks to check whether it violates the policies in the Policy List. It bridges the file-level and block-level semantic gap by designing a FSP (File System Parser) which is used to analyze Windows FAT32 file system.

However, these methods are operation-based, which implies file operation is intercepted by monitoring tools as long as it is being executed. Different with the existing works in the literature, our paper aims to design and implement a target-based low-overhead approach to monitor critical files effectively by using VMI, which can support both Linux and Windows VMs. Several papers (e.g., [24]–[29]) have also studied related security issues.

3. Design

3.1 Overview

To the best of our knowledge, most of existing real-time file monitoring approaches are based on operation. Due to the system-level interception on file operation, the majority that are irrelevant to critical files will waste too many processing capacities. Different from the operation-based schemes, our proposed solution in this paper is a target-based one that monitors merely the events of target files, which consequently reduces potential overhead dramatically.

Similar to most VMI-based systems, CFWacther harnesses VMI techniques to monitor VM filesystem as illustrated in Fig. 1. The right is a normal VM (target VM) to be monitored, and the left is the privileged VM (Secure VM) where CFWatcher works. With a higher privilege than VM, VMM can inspect the events in target VM and redirect them into Secure VM. Then, Secure VM can change the content of target VM's underlying hardwares if necessary. There are two major components: Policy Engine and Toolkit Library. Policy Engine manages security policies while Toolkit Library is responsible to interact with VMM and provide a holistic view of target VM for Policy Engine through interpreting hardware states exported by VMM. When an event is passed from Toolkit Library, Policy Engine will react with proper actions after analysis.

Aiming at target-based monitoring, we explore the internal mechanisms of Linux and Windows filesystems in depth. In general, one common file contains two kinds of information: real data (content) and meta data (file name, file type, file size etc). When opening a file, meta data will be loaded into the memory at first, and later the content will be retrieved at the request of read/write event. To optimize memory utilization, if some meta data already exist in the memory (for example, the file has been opened by other processes), OS will increment reference counter of meta data rather than loading another hard copy. In particular, when the counter is reduced to zero, meta data may be freed up for more available memory. According to the above observation, such reference counters are useful to monitor the files in a target-based mode, i.e. to monitor the events over the corresponding memory area. Considering the differences between Linux and Windows filesystems, we design distinct approaches to be accustomed to them, which will be detailed in the following parts respectively.

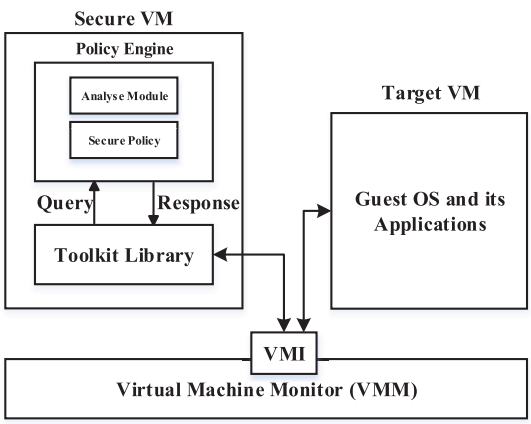


Fig. 1 The architecture of CFWatcher.

As a prerequisite, meta data have to be cached into the memory in advance, but it is not sure that target files had been accessed before or meta data are maintained all the time. Thus, two different methods of meta data construction are proposed in this paper to guarantee the liveness of meta data.

Besides, as long as the operations on target files are detected, CFWatcher immediately begins to examine the authorization according to security policies that users can predefine in a complex way. For example, if the file */test/test* is permitted to be opened by the user *admin* with the program *cat*, CFWatcher will restore the running state of VM and check whether the program is *cat* and the user is *admin* when */test/test* is accessed at runtime. If a violation occurs, CFWatcher will trigger an alarm and take actions to abort the operation.

We assume that the VMM and the SVM are both secure, and the guest VM may be attacked by attackers. However, since CFWatcher depends on the meta data reference counter mechanism, the subversion of this mechanism could affect CFWatcher. To the best of our knowledge, this kind of attack does not exist, but we should also assume that the reference counter mechanism should be integrity.

3.2 CFWatcher Monitor in Linux

All files are accessed through Virtual File System (VFS) [17] in Linux. VFS regards each directory as an ordinary file that consists of several sub-directories and files. As one directory or file is loaded into memory, the kernel will create a designated object named dentry. Generally, the kernel will create a dentry object for every part in the pathname. For example, when searching the path */home/target-file*, the kernel will first create a dentry for /, then create a second-level dentry for *home* beneath /, and finally create a third-level dentry for target-file beneath *home*. Since dentry creation is a time-consuming operation, dentry maintenance for future use becomes important to better memory efficiency even if previous operations have finished.

All dentry objects are stored in a special memory area called dentry cache. Due to limited memory space, the kernel needs to clean up unused dentry objects. In order to prevent in-use dentry objects from discarding, reference counter is designed to record how many processes are using the dentry object. When a process opens a file, the reference counter will be incremented. Otherwise, the counter will be reduced until zero if no process uses the file. As shown in Fig. 2, when another process accesses the same file whose dentry object has existed, it will increase the corresponding reference counter instead of creating a new dentry object.

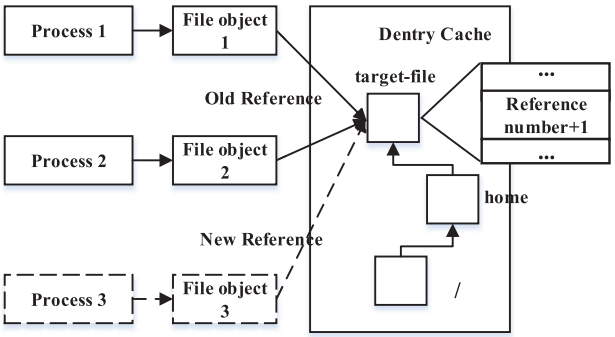


Fig. 2 Dentry and reference number.

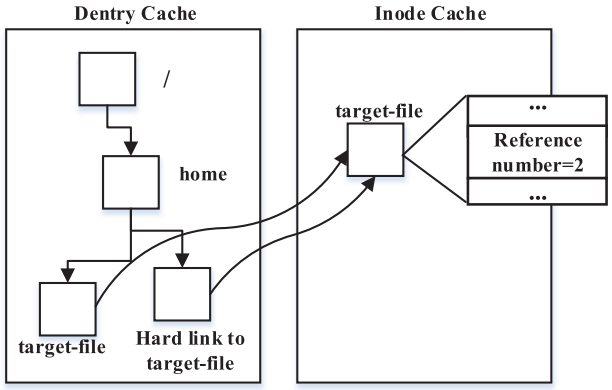


Fig. 3 The relationship between dentry and inode.

In our design, CFWatcher actually monitors the operations on target files via the dynamicity of reference counters used by dentry objects over time. It can be achieved in the following steps:

1. Locate the region of the dentry objects corresponding to target files in VM's memory with OS-level semantic information.
2. Find the memory address of the reference counter by calculating its offset in dentry structure.
3. Monitor the changes of these memory area. If some reference counter increases, which means the corresponding file is being accessed, CFWatcher can be aware of the events in real time so as to take necessary actions in time.

When no process uses a file, the reference counter of its dentry object will be decreased to zero and later the dentry object may be discarded. To maintain the monitored dentry objects alive in memory, CFWatcher will increase the reference counter before it becomes zero. In this case, even though there is no valid reference to the dentry object, the value of its reference counter is one and it will be alive in memory all the time.

Notably, a file is actually regarded as a dentry in view of VFS but as an inode in view of disk. That is, each dentry corresponds with one filepath while each file on the disk corresponds with one inode object. However, an inode object may be pointed to by multiple dentry objects. For example, hard link can make this happen because a file with hard links can have multiple paths as shown in Fig. 3. When this file is accessed, the corresponding inode and dentry objects will be loaded into the memory together. Fortunately, there is also a field in inode structure to record how many hard links point to itself. When a file is removed, the inode object's hard links will decrease correspondingly. As a result, our CFWatcher can succeed in detecting the deletion of target files by monitoring the number of inode objects' hard links, which similar to monitoring the dentry object's reference counter.

3.3 CFWatcher Monitor in Windows

To support better compatibility, we work on the NTFS filesystem of Windows 7. Because Windows is a private-proprietary OS and the memory layout of NTFS filesystem is unknown, the design in Windows is more challenging than in Linux. By analyzing the binary data in the memory deeply, CFWatcher restores part of semantic information to enable the monitor on NTFS filesystem.

Specifically, as every process opens a file in Windows, an object _FILE_OBJECT will be created and stored in the handle_table. If the file is opened more than one time, each operation will generate an independent _FILE_OBJECT that is stored in the handle_table affiliated to the process.

In NTFS, there is a unique File Control Block (_FCB) object stored in the memory, which corresponds to each opened file. Similar to the function of *inode* object in Linux, _FCB records the state of a file, such as hard links. A file can have many attributes, including an unnamed attribute and multiple named attributes, and every attribute can be opened and operated independently. Unnamed attribute is used to store file content while named attribute usually stores meta data (like author, version et al.). When operating an attribute, a corresponding Stream Control Block (_SCB) object will be created in the memory. These objects are organized together as a bi-direction link. All _FILE_OBJECT objects in a process point to the corresponding _SCB object. In summary, there is only one _FCB object for a file, but this object can be in correspondence with multiple _SCB objects. Every _SCB object may be referenced by multiple _FILE_OBJECT objects. Like *dentry* object in Linux, _SCB object records how many _FILE_OBJECT objects are referencing on itself so as to stay alive in the memory.

As shown in Fig. 4, *process 1* opens a named attribute and an unnamed attribute simultaneously, which correspond with two different _SCB objects respectively. Also, they both point to a single _FCB object. If a new *process 2* wants to access the content of this file, the _FILE_OBJECT object will point to the _SCB object and the reference counter in _SCB object will be increased. Similarly, CFWatcher monitors the file by sensing the change of reference counter in the corresponding _SCB object. An increase of reference counter means that the file is being accessed.

CFWatcher in Windows works very similar with Linux.

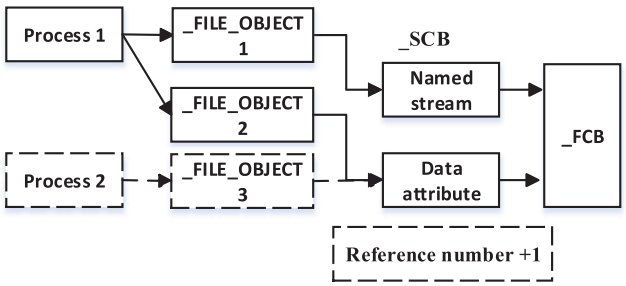


Fig. 4 Core structures of NTFS filesystem.

The main difference is using _SCB object in Windows instead of dentry object in Linux.

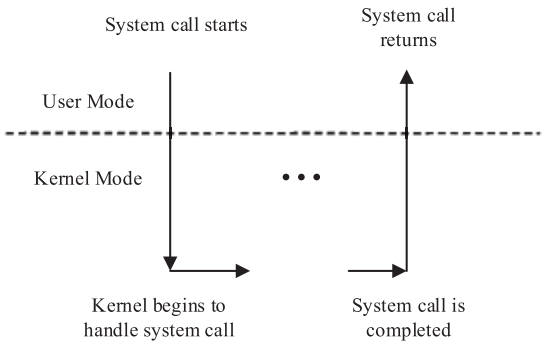
3.4 Creating Cache Objects

To ensure the existence of necessary caches in memory before monitoring, we design two cache construction approaches, i.e. system-call-based and agent-based, which are both suited for Linux and Windows. In a modern OS, all important resources (such as files, devices) can only be accessed via system calls. The objects in previous sections (dentry and _SCB) actually are all set up by invoking particular system calls. Therefore, we can use this feature to open target files and construct their caches in memory.

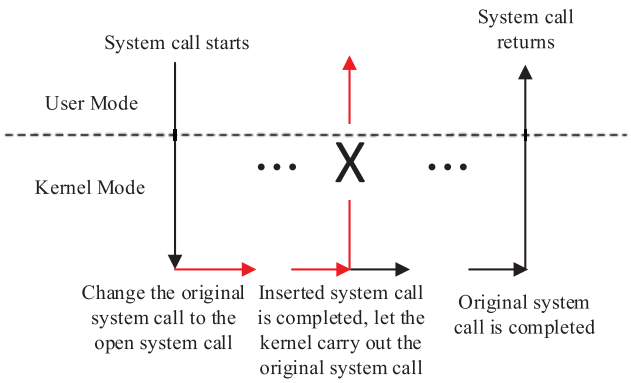
As a completely out-of-box solution, the core idea of system-call-based approach is to insert an extra open system call into on-the-fly normal routine, which will open target files to be monitored. Figure 5 (a) shows the normal routine where the OS will enter kernel mode so as to handle the system call invoked by users. As shown in Fig. 5(b), we monitor the events on the execution of VM's system calls outside the VM. As long as target VM switches into kernel mode, the original system call will be modified immediately and then redirected to our extra open system call. After finishing our add-on system call, we resume the original system call by manipulating instruction pointer. In this case, original system call can be continued and will return as expected, which is totally transparent to users.

Agent-based approach is simpler than system-call-based one. An agent runs in a VM and is responsible to open all of our monitored files, which can be launched as a daemon process at boot time. Through memory sharing mechanism with VMM, CFWatcher is able to store the paths of target files into the memory of the agent. After the agent finishes all opening operations, the states of target files can be fed back in the same way. At last the agent will shut down automatically.

Actually, it is hard to tell which approach is better than the other one. System-call-based approach is completely transparent to users and more secure, but it manipulates instruction pointer, which brings unreliability to commercial systems. In the contrary, agent-based approach is more reliable and easy to deploy, but it may be detected and even compromised by malwares.



(a) Without injection.



(b) With injection.

Fig. 5 System call routines.

3.5 Taking Actions

CFWatcher is triggered by intercepting the operations on target files, which can enable timely and proper actions against malicious intentions. To prevent the unauthorized access, CFWatcher makes ongoing system calls fail by setting their return values to a negative number and clearing the related file descriptors, which does not allow the process to access the target file as usual. Clearing the related file descriptor is a must after setting return value, because it is possible for a process to guess potential locations of file handlers according to memory layout even though open operation is failed. Moreover, we also need to protect key objects (such as *inode* or _FCB) from modification, which can prevent the removal of target files.

4. Implementation

4.1 Environment

CFWatcher prototype is built on an x86 server with Intel VT [18] technique support. It can also be implemented on 64-bit platforms. The VMM is a popular open-source hypervisor - Xen [19], which is widely used in cloud computing. The VMs running atop VMM are Ubuntu and Windows 7 distributions.

Since all data in VM's memory are binary mode in-

stead of meaningful semantic context in the VMM's view, VMI-based tools have to restore semantic information from raw binary data, which is so-called semantic gap. Fortunately, there are many efforts contributed by memory analysis communities who template most of kernel data structures (covering the majority of Linux and Windows distributions) by analyzing binary data in memory. Therefore, we leverage the Volatility Framework [20] to provide an OS-level view of guest VM and extract semantics automatically in our prototype. As a well-known memory forensics framework, Volatility integrates and builds the set of memory templates with which it can automatically extract semantic context from binary data. Because Volatility is a programmed OS-level analysis toolkit, CFWatcher is able to monitor most of operating systems without any modification. Using Xen toolkit library to expose the memory of a VM to the Dom0, we can read the live VM's memory to allow Volatility to analyze the binary data directly.

4.2 Initializing Cache Objects

In the Sect. 3 we introduce two approaches (i.e. System-call-based and Agent-based) to create cache objects if they are not alive in memory. Here, we emphasize on system-call-based one as a case. The core idea of system-call-based method is to insert an extra open into normal routine when system call is invoked. To that end, we exploit the features of Intel fast system call entry mechanism in Linux 2.6, Windows XP and later.

In Intel x86 architecture, SYSENTER/SYSEXIT [21] instructions are used for fast entry to switch between kernel and user mode. SYSEXIT is a companion instruction to SYSENTER. The SYSENTER instruction is used by system calls to convert user mode to kernel mode. Meanwhile, EAX register stores the number of system call, EBX stores the first argument, and ECX stores the second argument, and so on. SYSEXIT leverages EAX to pass return value when system call is finished. It is easy to modify the number and parameters of system call by changing the content of relevant registers when SYSENTER/SYSEXIT is executing, which is exploited by CFWatcher as follows:

1. It monitors the operation on the kernel point address, i.e. the execution of the kernel entrance instruction, whose address is stored in the register IA32_SYSENTER_EIP.
2. When SYSENTER is in execution, it stores the content of all registers at first, sets RAX register to system call number of our extra open, and replaces other registers with the corresponding parameters.
3. It captures the execution of kernel entrance whose address is not far away from the kernel point and can be obtained by traversing the kernel's memory.
4. After finishing our extra system call that will open all target files stealthily, IP register will be reset to the kernel point address and meanwhile other registers will also be recovered with previous parameters stored in Step 2. Especially, we need to carefully tackle the correspondence between SYSENTER and SYSEXIT instructions. Since a process may be switched out in the context, an SYSENTER followed by an SYSEXIT may be generated by different processes. Because a process can not request a new system call before the previous one has not finished yet, CR3 register can be used to map SYSENTER and SYSEXIT with each other. All threads in a process have the same CR3 value, but they have different ESP values. Therefore, when CFWatcher captures SYSEXIT instruction, it will compare the CR3 value and the ESP value with those of SYSENTER in Step 2.

4.3 Locating Cache Objects

Before monitoring, the cache objects of target files must be located in VM's memory. Because Volatility allows developers to easily extend new functions for different uses, we design a new plugin called *cachefinder* to find out the related objects of target files in memory, i.e. dentry/inode in Linux and _SCB/_FCB in Windows. Besides, cachefinder will record all addresses of target files so as to facilitate the monitoring.

Cachefinder utilizes the existing templates for the extraction of kernel data structures. The templates in Volatility that contain dentry/inode structures is sufficient for Linux to find cache objects in VM, but more efforts are required in Windows because _SCB and _FCB structures are still unknown except _FILE_OBJECT. Through binary analysis, we can find the reference to _SCB, the pointer to _FCB and the offset of hard link's counter in _FCB structure.

4.4 Monitoring Special Memory Region

CFWatcher monitors the modification and execution of special memory region using Intel Extended Page Table (EPT) mechanism that is used to provide memory virtualization. The EPT's permissions deliberately granted by us can help CFWatcher to realize whether any special EPT page is being accessed.

Basically, CFWatcher sets the EPT entries of physical pages related with monitored memory region to read-only (to intercept the write events) or read-only and write-only (to intercept the execution events) [22]. When a target page is accessed, an EPT violation will be triggered and then captured by CFWatcher. If the accessed address lies in the region to be monitored, which means some of target files are being accessed, CFWatcher can invoke the predefined handler. Then, CFWatcher sets the corresponding EPT entries to normal and resumes target VM in single-step mode, so that target VM can access the special memory region. After finishing the operations, CFWatcher finally resets the corresponding EPT entry to read-only for future events.

4.5 Getting Target Process Information

To support advanced access rules, CFWatcher extracts the semantics of the process that is operating the files. Every process in Linux has a structure named task_struct that maintains several related information, such as PID, UID, GROUPID et al. All task_struct data are linked in a bi-direction way and start from the address of init_task. Similar to Linux, the process information in Windows is stored in the _EPROCESS structure. CFWatcher locates the corresponding structure of target process via CR3, and then leverages Volatility to restore the semantics of the process automatically.

4.6 Preventing Rule Violations

CFWatcher modifies the illegal system call's return value to prevent rule violations. After checking rules, CFWatcher monitors the execution of the illegal system call. When the corresponding SYSEXIT is executed, CFWatcher sets the value of EAX to -1 and clears the related file descriptors.

5. Evaluation

In order to demonstrate CFWatcher's advantages, we evaluate our prototype system in a testbed, which runs in a hardware-assisted virtual environment - Xen 4.3 with the configuration of 2.4GHz Intel Core i5 dual-core processor and 4GB memory. Meanwhile, two VMs to be monitored, i.e, 32-bit Ubuntu 12.04 and 32-bit Windows 7, are hosted in a full-virtualization mode. In the following we will demonstrate the effectiveness and performance in Linux and Windows VMs respectively.

5.1 Effectiveness

5.1.1 Linux

Rootkits are very common in Linux and mainly compromise the kernel in two different ways: directly modifying the kernel's memory or inserting themselves as LKM modules. In our experiment Phalanx is employed as a testing tool, which is a self-injecting kernel rootkit designed for Linux 2.6 branches. It uses */dev/mem* interface to inject hostile codes into kernel memory and hijack system calls. Moreover, to enable the compatibility with our testbed (Ubuntu 12.04 with kernel version 3.2), we rewrite Phalanx and render it runnable in target VMs. The whole processing is performed as follows: 1) initialize CFWatcher and configure it to monitor the file */dev/mem*; 2) run Phalanx to modify the kernel. Our experimental results show that CFWatcher can detect and prevent the operation on */dev/mem* triggered by Phalanx, which proves its effectiveness in Linux.

5.1.2 Windows

In this part, system critical file Host is used to evaluate the effectiveness in Windows, which is actually an associated database that stores the mapping between common domain names and their ip addresses. When a user inputs a domain name in the browser, the system first automatically confirms whether the corresponding ip address exists in the Host. Thus, Host is often able to speed DNS resolving or block some websites locally. However, malwares misuses this feature to modify the Host and then compromise the users, such as logging on a phishing website. Similarly, we write a program to simulate an attack on the Host.

In the test, we first initialize CFWatcher to monitor the Host. Then, our program attempts to modify the Host. The experimental results show that CFWatcher can capture and prevent the operation on the Host when the modification occurs, which proves the effectiveness of CFWatcher in Windows.

5.2 Filesystem Performance

5.2.1 Linux

As a filesystem monitor, CFWatcher definitely involves in additional overhead on VM filesystem, which should be estimated, especially performance variation with the increasing number of target files. Thus, we use a build-in command *dd* to assess the overhead of VM filesystem. *dd* is a command-line utility whose primary purpose is to convert and copy files in Unix and Unix-like OSes. To simulate the workloads, we make use of device drivers for hardware (such as hard disks) and special device files (such as */dev/zero* and */dev/null*) that *dd* can read/write. */dev/zero* can provide as many null characters (ASCII NUL, 0x00) as requested, so copying data from it can generate write workloads exclusively. In the contrast, */dev/null* is a device file that discards all data written to it, which can barely produce read workloads by copying data to it. In short, *dd* can be used to generate and measure the read/write operations. Different from other file access controllers, CFWatcher is sensitive to read/write operations. When lots of files are monitored, CFWatcher monitors a large scale of memory pages of the VM file cache. However, reading and writing files will also cause memory operations in this area. As a result, if we monitor lots of files, the monitored pages are usually accessed by these file operations, causing massive overhead. Since CFWatcher also impose overhead on open/close operations, we will evaluate them in another benchmark.

In our test we measure read speed by using the command *dd bs=128k count=10240 if=/dev/xvda of=/dev/null*, which copies 10K blocks (128KB) from the disk to */dev/null*. Then we use *dd bs=128k count=10240 if=/dev/zero of=testfile* to measure write speed. We compare the read/write performance between original VM and CFWatcher-enabled VM. Figure 6 shows the read/write performance of the VM when CFWatcher monitors a varying number of files. The horizontal axis represents the number of files to be monitored (0 means that CFWatcher is disabled), and the vertical axis represents the read/write speed

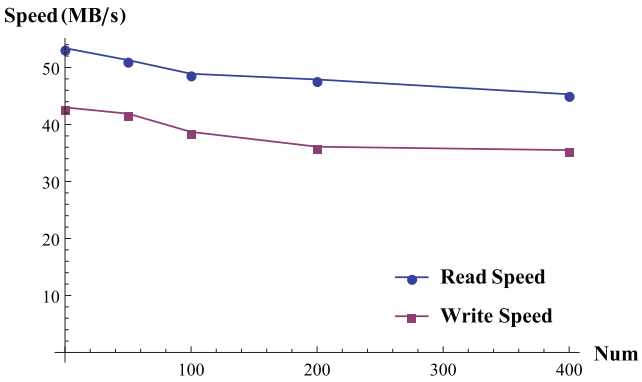


Fig. 6 Linux filesystem performance.

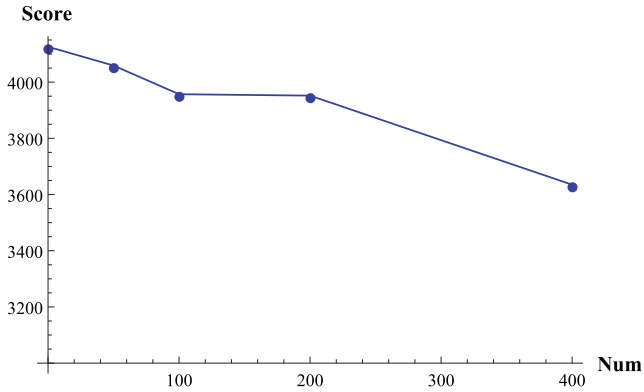


Fig. 7 Windows performance.

Table 1 Processing time of unzipping kernel.

Monitored Files	Time
0	25
50	25.3
100	26.1
200	27.1
400	29.8

(MB/s). Empirically, the number of in-use files in a running Linux is 900 on average. Our tests monitor no more than 400 files simultaneously.

From Fig. 6 we can see that when the number of monitored file is less, the loss of read/write performance will be smaller. For 50 monitored files, the performance loss as compared to an unmonitored system is less than 5%. And in the case of 100 monitored files, the performance loss is less 9%. The max read performance decrease is 15% while the max performance decrease is 17.4% when the number of monitored files is up to 400.

Besides read/write speed, other file operations like creation and copy, are also important to filesystem performance. Thus, we employ compression tools to unzip a Linux kernel, which can create a large number of small files in a short time, to evaluate the influence on filesystem performance. We compare processing time when monitoring different numbers of files (0 means that CFWatcher is disabled).

The results are shown in Table 1: when the number of monitored files is less than 100, time difference is less than 5%; when the number is up to 400, time difference also increases to 19.2%.

In the above test cases, the files accessed by the benchmark programs are not included in the monitored files. According to our design, when monitored files are being accessed, rule checker will be triggered immediately, which also brings extra overhead. Thus, we write a program deployed in the monitored VM to open 100 monitored files successively and record total time spent. The experimental results show that rule checker introduces extra 2ms. If the monitored files are not accessed continuously, rule checker will only have a small impact on the whole system. Furthermore, the performance of rule checker could be optimized. The overhead is mainly introduced by getting the target process information, which is completed in real time now. If we cache the obtained information and update it periodically, the overhead will be significantly decreased.

5.2.2 Windows

To evaluate the overhead in Windows VM, we make use of PCMark 7 [23], which is a complete PC benchmarking solution, including 7 tests combining more than 25 individual workloads covering storage, computation, image and video manipulation, web browsing and gaming. In our test we run storage suite in PCMark 7 to evaluate filesystem performance. The storage suite is a collection of workloads that isolate the performance of PC's HDD or SSD, such as Windows Defender, importing pictures, video editing. It also can test other storage devices in addition to the main system drive. After finishing the test, PCMark will give a performance score.

We run storage suite in different VMs: CFWatcher-disabled, CFWatcher with monitoring from 50 to 400 files. Figure 7 shows that performance loss is similar with Linux VM, but is better than Linux VM. When monitoring 50 files simultaneously, the loss of filesystem performance is 1.6%. If the number of files is up to 100, the loss will increase to 4.9%. At the worst, when CFWatcher monitors 400 files at the same time, the performance of VM filesystem become worse by 11.9%. In the above test cases, the files accessed by the benchmark programs are not included in the monitored files. We also measure the overhead introduced by rule checker when monitored files are being accessed. Through the test, the overhead of rule checker is also 2ms.

5.3 Memory Performance

Since we employ Intel EPT to set up the monitoring, we need to evaluate the influence on memory performance via Memory Bandwidth Benchmark (MBW) tools. Bandwidth [24] is a benchmark primarily for measuring memory bandwidth on x86 and x86_64 based computers, useful for identifying bottlenecks in memory subsystem. It not only

Table 2 The comparison of CFWatcher and operation-based approach.

Benchmarks	Operation-based		Target-based	
	Performance	Overhead	Performance	Overhead
Reading speed	42.4MB/s	20.5%	45.3MB/s	15.2%
Writing speed	32.3MB/s	25%	35.5MB/s	17.4%
Unzipping Linux kernel	78s	212%	29.8s	19.2%

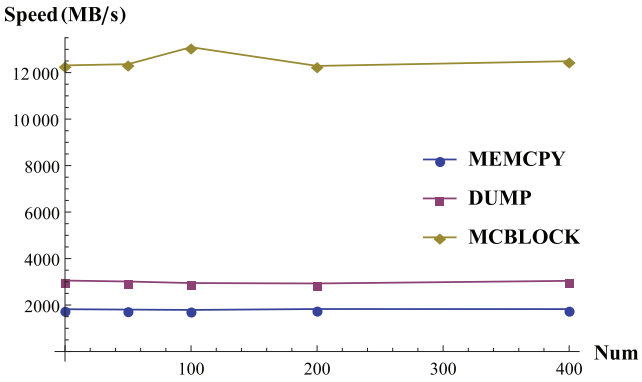


Fig. 8 Linux memory performance.

can read/write data blocks of different size in the sequential or random way, but also can simulate the read/write patterns of normal softwares.

In our experiments we run the command *mbw 128*, which performs bandwidth measurement on 128M memory in MEMCPY, DUMB and MCBLOCK modes. We compare the memory bandwidth with/without CFWatcher as shown in Fig. 8 where the horizontal axis means the number of monitored files (0 means that CFWatcher is disabled), and the vertical axis means memory bandwidth.

We can see that the memory bandwidth is not always the highest when CFWatcher does not work. The variation of all results is no more than 6%, which implies that the influence of CFWatcher on memory performance is small. Because we use Intel EPT to enable the monitoring on VM memory region in VMM independent on OS types, it is easily inferred that the overhead of Windows VM is equivalent to Linux VM.

5.4 Comparison with Operation-Based Approaches

5.4.1 Linux

To compare with operation-based approaches, we also implement an operation-based approach in Linux VM. We monitor the execution of the kernel entrance instruction to intercept the system calls. We first set the corresponding memory page to non-executable. When the page is executed and the execution address is the kernel entrance, we can aware that a system call is executed in the monitored VM. By using this method, we monitor several system calls (i.e. sys_open, sys_read, sys_write), which are usually monitored by operation-based file monitors (e.g., [7]) to intercept file operations. To simplify the experiment, we record system call information when intercepting file operations. We use the same benchmarks (reading/writing speed and unzipping Linux kernel) to evaluate the performance in this condition.

The experimental environment is as the same as CFWatcher's, and the results are shown in Table 2. From the results, we can see that when even monitoring 400 files, the performance of CFWatcher is better than that of operation-based approaches.

5.4.2 Windows

We also implement an operation-based Windows filesystem monitor in VMM to compare with CFWatcher. When target system calls are intercepted, we get the arguments by reading registers, and then log them. We use PCMark 7 storage suite to evaluate the performance of operation-based monitor. We run ten times to calculate the average score. From the results, performance loss is about 35%, which is much larger than CFWatcher.

5.5 Comparison with the Commercial Approach

We also compare CFWatcher with commercial approach. VMware provides an agentless security solution named vShield Endpoint, which offloads antivirus agent processing to a dedicated secure virtual appliance delivered by VMware partners. By using a thin agent installed in the monitored VM, vShield Endpoint can monitor VM events and notify the antivirus engine. Trend Micro Deep Security can leverage vShield Endpoint to scan on-access files in VM. To compare with it, we deploy vShield Endpoint on VMware ESXi 5.5 and VMware vCenter 5.5. Then we use Trend Micro Deep Security 9.6 as the antivirus engine. Since vShield Endpoint only supports Windows VMs, we compare the performance of CFWatcher with that of VMware solution in Windows VM. PCMark 7 storage suite is also selected as the benchmark. We first run the benchmark process in the VM with vShield enabled, and then run it in another VM with vShield disabled. The experimental results show that the average performance loss is about 11.5%, which is larger than that of CFWatcher with less than 385 monitored files in Windows VM.

5.6 Discussion

According to experimental results, we can conclude that no matter what OS the VM is, the impact on VM performance will become more serious if the number of monitored files increases. The reason is: monitoring memory region works at page level. When we need to monitor a special memory

region in a page, we have to set the page to read-only or write-only mode. At the same time the modification on other data in the same memory region inevitably will bring extra overhead. The more files are monitored, the more pages will be monitored, which consequently increases the overhead. Through our tests, when the number of monitored files is small, specially no more than 50, the overhead is very low.

There is another interesting observation that the performance of Window VM is better than that of Linux VM. The reason is that file control structure in Linux is more intensive, which implies that more structures coexist in one page. When one of them is monitored, the access to others that should not be monitored in the same page generates extra overhead and has a serious impact on the performance. By comparison, we find that the distribution of file control structures in memory is more sparse, the overhead is lower.

We can also find that our operation-based method's overhead is much larger than other system call interception methods, such as [7] (10.7% in the Linux case). We think the main reason is that our interception method is at page-level, which means that the execution of every instruction in the monitored page will trigger the VMExit event, causing a heavy load. As shown in Sect. 5.2.1 and 5.2.2, the overhead of CFWatcher varies with the number of the monitored files. But the performance of operation-based approaches is not related to the number of monitored files. So even comparing with the overheads reported in other related studies, the overhead of CFWatcher will be smaller in the case of monitoring a small number of files (e.g., less than 100 files).

Cache object monitoring of CFWatcher may cause some other effects, such as memory shortage. When the user specifies a large number of files as critical, there are lots of cache objects occupying the VM memory. Since these objects are small in size (e.g., the dentry object is 128 bytes in size), monitoring hundreds or thousands of files can not lead to memory shortage. However, if the user monitors tens of thousands of files, that will cause memory shortage. Furthermore, the performance degradation on filesystem will also be very high in this case. So CFWatcher is more suitable for monitoring a small number of files. We have also tested the effect on VM shutdown, because the monitored files can't be closed. In the test, the VMs can be shutdown properly with several files monitored.

6. Security Analysis and Countermeasures

An attacker may exploit some vulnerabilities to bypass CFWatcher, and even try to compromise CFWatcher especially when detecting its existence. In this part, we initial the discussion on security analysis, including potential attack vectors and countermeasures.

6.1 Hard Link

Hard link mechanism in both Linux and Windows allows a file to have multiple different filepaths, which is exploited to circumvent most of malware detection based on the filepath. An attacker first creates a hard link or uses an existing hard link to connect with one monitored file. As a result, there are two filepaths in the filesystem both pointing to the same monitored file. If we monitor the paths instead of the hard links, the attacker can easily access the monitored files with different paths so as to bypass the monitor.

Fortunately, hard links related to the same file have their own dentry objects in the memory, which are organized as a bi-direction link. In order to handle with hard links, CFWatcher will search all dentry objects related to target files if hard links exist. Consequently, CFWatcher can fight against the attacks on hard links by examining the change of their number when connecting or disconnecting to target files.

6.2 Direct Operations

In fact CFWatcher is built on the filesystem of modern operating system to monitor target files. Based on this knowledge, an attacker may circumvent CFWatcher by directly operating storage devices. For example, he can read the binary data from the underlying disk and afterwards reassemble the meta data. That way, the attacker can locate the storage addresses of target files on the disk successfully, which allows the attacker to access target files at last. Since the attacker does not operate target files using filesystem functions, there are no corresponding objects that are available for CFWatcher to monitor the operations on target files.

The basic idea of above attack is to directly read/write the underlying devices. They are similar in both Linux VFS and Windows NTFS through accessing special device files, i.e. */dev/xdva* is the path of hard disk in Ubuntu while *D:* is the path of logical D disk in Windows. In order to fight against this attack, CFWatcher can enable the monitor on read/write events of these files.

6.3 Kernel Attack

Kernel attack is another way to subvert CFWatcher by tampering or even replacing the kernel. CFWatcher is based on the meta data reference counter mechanism, CFWatcher can not work well if this mechanism is compromised. This mechanism is very difficult to be tampered, because it is a core mechanism of filesystems. As a result, there are no rootkits attacking it. To eliminate this risk completely, we can employ existing kernel protection tools to defeat it.

7. Conclusion

In this paper we present CFWatcher, a target-based real-time monitoring approach for critical files. Different from operation-based solutions, the overhead introduced by CFWatcher depends on the frequency of target files being accessed rather than the whole filesystem. The core idea is to constrain the interception on file operations into critical files, which consequently reduces the overhead. In order

to facilitate target-based monitoring, we propose two construction methods (i.e. system-call-based and agent-based) to set up associated entities corresponding to target files in VM's memory. By monitoring on in-memory meta information, CFWatcher can prevent unauthorized access in an out-the-box way once any invalid operation is detected. Furthermore, we extensively investigate the differences between Linux and Windows to render CFWatcher more feasible in practice. With the prototype system built on Xen, we evaluate the effectiveness and performance in Linux and Windows VMs respectively. Through the experiments, the overall overhead is acceptable. Specifically, when the number of target files is below 50, it is less than 5%; and in the case of 100 target files, it is less than 9%.

In the future, we are going to extend our system to a full-featured IDS and improve the performance with the increasing number of target files.

Acknowledgements

This work was partially supported by the program of China Scholarship Council (CSC), the Enterprise-University-Research Institute Cooperation Project of Guangdong Province, China under grants NO. 2016B090921001, and National Natural Science Foundation of China under grants NO. 61601146.

References

[1] C.A. Adukkathayar, G.S. Krishnan, and G. Sasikumar, "Advanced integrity checking and recovery using write-protected storage for enhancing operating system security," Computer Science & Education (ICCSE), 2015 10th International Conference on, pp.219–224, IEEE, 2015.
[2] J. Kaczmarek and M.R. Wrobel, "Operating system security by integrity checking and recovery using write-protected storage," IET Information Security, vol.8, no.2, pp.122–131, 2014.
[3] N.A. Quynh and Y. Takefuji, "A real-time integrity monitor for xen virtual machine," Networking and Services, 2006. ICNS'06. International conference on, p.90, IEEE, 2006.
[4] Z.H. Abdullah, N.I. Udzir, R. Mahmod, and K. Samsudin, "File integrity monitor scheduling based on file security level classification," International Conference on Software Engineering and Computer Systems, vol.180, pp.177–189, Springer, 2011.
[5] S. Gupta, A. Sardana, and P. Kumar, "A light weight centralized file monitoring approach for securing files in cloud environment," 2012 International Conference for Internet Technology And Secured Transactions, pp.382–387, IEEE, 2012.
[6] R.K. Ko, P. Jagadpramana, and B.S. Lee, "Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments," 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, pp.765–771, IEEE, 2011.
[7] H. Jin, G. Xiang, D. Zou, F. Zhao, M. Li, and C. Yu, "A guest-transparent file integrity monitoring method in virtualization environment," Comput. Math. Appl., vol.60, no.2, pp.256–266, 2010.
[8] N.A. Quynh and Y. Takefuji, "A novel approach for a file-system integrity monitor tool of xen virtual machine," Proceedings of the 2nd ACM symposium on Information, computer and communications security, pp.194–202, ACM, 2007.
[9] N. Li, B. Li, J. Li, T. Wo, and J. Huai, "vmon: an efficient out-of-vm process monitor for virtual machines," High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, pp.1366–1373, IEEE, 2013.
[10] F. Liu, H. Zhang, and H. Zhou, "A xen-based secure virtual disk access-control method," 2010 International Conference on Multimedia Information Networking and Security. Nanjing, Jiangsu China: IEEE Computer Society, pp.375–378, 2010.
[11] J. Wang, M. Yu, B. Li, Z. Qi, and H. Guan, "Hypervisor-based protection of sensitive files in a compromised system," Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp.1765–1770, ACM, 2012.
[12] D. Zhan, L. Ye, B. Fang, X. Du, and S. Su, "Cfwatcher: A novel target-based real-time approach to monitor critical files using vmi," Communications (ICC), 2016 IEEE International Conference on, pp.1–6, IEEE, 2016.
[13] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," NDSS, pp.191–206, 2003.
[14] J. Hizver and T.-c. Chiueh, "Real-time deep virtual machine introspection and its applications," Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp.3–14, ACM, 2014.
[15] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp.839–850, ACM, 2013.
[16] H. wook Baek, A. Srivastava, and J. Van der Merwe, "Cloudvmi: Virtual machine introspection as a cloud service," Cloud Engineering (IC2E), 2014 IEEE International Conference on, pp.153–158, IEEE, 2014.
[17] "Vfs," http://wiki.osdev.org/VFS
[18] Intel Inc., "Intel® 64 and ia-32 architectures software developer's manual," Volume 3A: System Programming Guide, Part, vol.1, p.64, 2013.
[19] "The xen project," http://www.xenproject.org/
[20] "The volatility foundation," http://www.volatilityfoundation.org/
[21] "Sysenter." http://wiki.osdev.org/SYSENTER
[22] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," Proceedings of the 29th Annual Computer Security Applications Conference, pp.289–298, ACM, 2013.
[23] "Pcmark7," https://www.futuremark.com/benchmarks/pcmark7
[24] "Bandwidth," http://zsmith.co/bandwidth.html
[25] Y. Cheng, X. Fu, X. Du, B. Luo, and M. Guizani, "A lightweight live memory forensic approach based on hardware virtualization," Elsevier Information Sciences, vol.379, pp.23–41, 2017.
[26] X. Fu, X. Du, and B. Luo, "Data correlation-based analysis method for automatic memory forensics," Security and Communication Networks, vol.8, no.18, pp.4213–4226, Dec. 2015. DOI: 10.1002/sec.1337
[27] X. Du, M. Rozenblit, and M. Shayman, "Implementation and performance analysis of SNMP on a TLS/TCP base," Proc. 7th IFIP/IEEE International Symposium on Integrated Network Management (IM 2001), pp.453–466, Seattle, WA, May 2001.
[28] Y. Xiao, H. H. Chen, X. Du, and M. Guizani, "Stream-based cipher feedback mode in wireless error channel," IEEE Transactions on Wireless Communications, vol.8, no.2, pp.662–666, Feb. 2009.
[29] X. Yao, X. Han, X. Du, and X. Zhou, "A lightweight multicast authentication mechanism for small scale IoT applications," IEEE Sensors Journal, vol.13, no.10, pp.3693–3701, Oct. 2013.

Dongyang Zhan received the B.S. degree in Computer Science from Harbin Institute of Technology (HIT), Harbin, China, from 2010 to 2014. From 2015 to now, he has been work as a Ph.D. candidate in Department of Computer Science and Technology at HIT. His research interests include cloud computing and security.

Lin Ye received his BSc, MSc, and Ph.D. degrees in Computer Science from Harbin Institute of Technology (HIT), Harbin, China, from 2000 to 2011. In 2012, he was a visiting scholar at Temple University, Philadelphia, PA, USA. He is currently a Lecturer in School of Computer Science and Technology in HIT. His research interests include peer-to-peer measurement, cloud computing and information security.

Binxing Fang is Member of Chinese Academy of Engineering. His research interests include Internet routing and security, wireless sensor networks, Internet of Things, Named Data Networking. He received Ph.D. (1989) and B.E. (1981) from Harbin Institute of Technology, and received M.E. from Qinghua University (1984).

Xiaojiang Du is a full professor in the Department of Computer and Information Sciences at Temple University, Philadelphia, USA. Dr. Du received his B.S. and M.S. degree in electrical engineering from Tsinghua University, Beijing, China in 1996 and 1998, respectively. He received his M.S. and Ph.D. degree in electrical engineering from the University of Maryland College Park in 2002 and 2003, respectively. His research interests are wireless networks, security, and systems. He has authored over 200 journal and conference papers in these areas. Dr. Du has been awarded more than $5 million US dollars research grants.

Zhikai Xu received the B.S. degree in Computer science from Dalian University of Technology (DUT), Dalian, China in 2009, and his M.S. degree in Computer Science form Harbin Institute of Technology, Harbin, China in 2011. From 2011 to now, he has been work as a Ph.D. candidate in Department of Computer Science and Technology at Harbin Institute of Technology, Harbin, China. His research interests include Mobile Cloud computing and security.