Inter-domain Communication Protocol for Real-time File Access Monitor of Virtual Machine

Ruo Ando*
National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795 Japan
ruo@nict.go.jp

Kazushi Takahashi
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
arena431@gmail.com

Kuniyasu Suzaki
National Institute of Advanced Industrial Science and Technology
1-1-1 Umezono Central-2, Tsukuba, Ibaraki, 305-8568, Japan
k.suzaki@aist.go.jp

Abstract

Leveraging hypervisor for security purpose such as malware analysis has been well researched. There still remain two challenges for analyzing security incidents on virtual machine: real-time monitoring and semantic gap. First, current active monitoring methods need to be improved for real-time protection of virtual machine. Second, semantic gap between virtual machine and hypervisor poses a significant impediment on security analyst. In this paper, we propose an inter-domain communication protocol for real-time monitoring of virtual machine and bridging semantic gap. We have deployed the inter-domain communication module between a guest Windows OS and a hypervisor in two ways. While the one is a register based transfer using vCPU context, the other is a shared memory based communication. Our protocol is event driven, which makes the proposed system enable to monitor the file access of a guest Windows OS in real-time without suspending it. We have implemented our system on XEN virtual machine monitor and KVM (Kernel Virtual Machine). We have measured the resource utilization of these two systems in the case of decompressing files and receiving HTTP requests. On the guest OS, the KVM based system outperforms the processor idle time by about 30-50% in decompressing file and the memory usage by about 35% in receiving HTTP requests. We conclude that our system can monitor file access inside virtual machine without suspension and also with reasonable resource usage.

Keywords: Virtual machine monitoring, inter-domain communication, file system driver, Xen and KVM

1 Introduction

Real-time Virtual Machine (VM) monitoring is now necessary and challenging problem for making ubiquitous virtual machines secure. With the rapid advance of hypervisor such as Xen, Linux KVM and VMWare, VM monitoring technologies have been well developed and researched. However, real-time access monitoring has not been proposed and is still not available, particularly in virtualized Windows

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 3, number: 1/2, pp. 120-137
*Corresponding author: Tel: +81-42-327-5344, Fax: +81-42-327-6634

OS. In other words, whereas VM status capture and analysis (for example, Volatility) is available, modules are not accessible to monitor resource access on real-time. There are two reasons for the difficulty of real-time monitoring. First, semantic gap between VM and hypervisor makes it hard to obtain log from outside the guest OS. Although we can intercept fine-grained events such as assembler instruction and I/O request, it is extremely hard to retrieve information from these events. Second, no inter-domain communication protocol and implementation has been proposed. Device driver specified for hypervisor has been pervasive, and split device driver to multiplexing and scheduling I/O request is proposed. However, there has been the lack of the inter-domain communication protocol specified for sending access log and its implementation. For providing real-time monitoring, we propose an inter-domain communication protocol for real-time malicious file access monitor of virtual machine. With our inter-domain communication protocol, the proposed system enables real-time monitoring with reasonable overhead less than 5%.

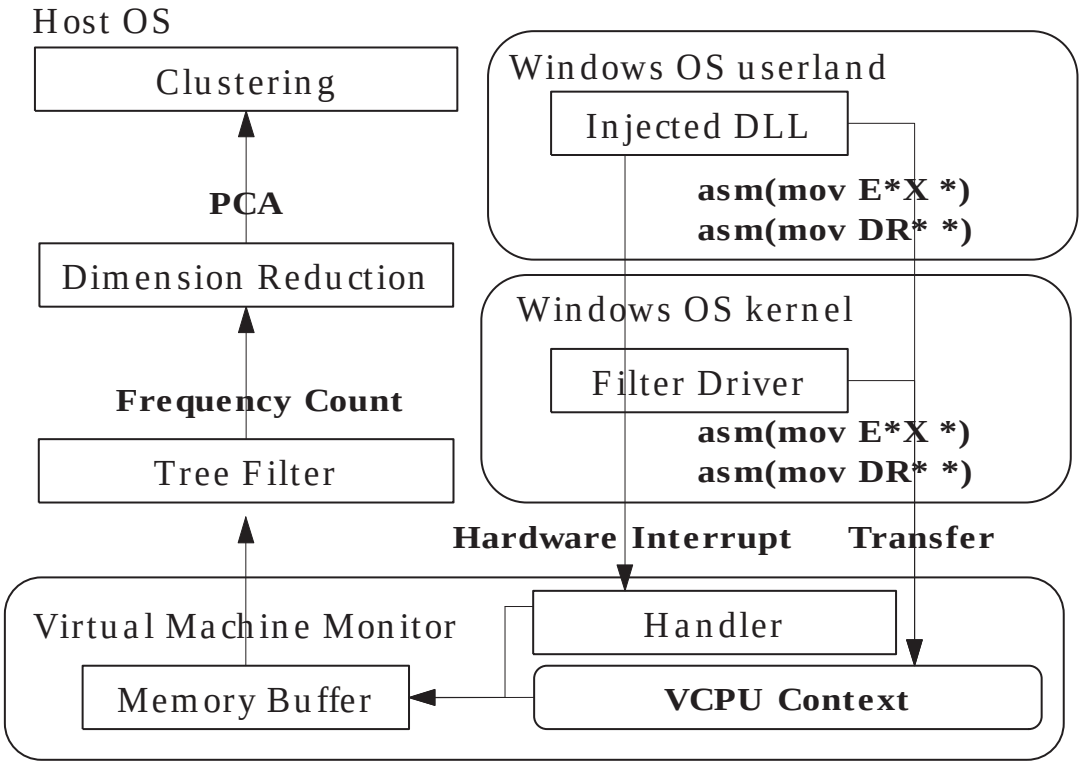


Figure 1: Proposed architecture for inter-domain communication.

2 Related work

Virtual machine monitor which is also called as hypervisor is a thin software multiplexing VM execution. Xen [1] and KVM [2] are open source software. Also VMWare ESX [3] and Windows 7 with XP mode [4] are commercial software of hypervisor. XenAccess [5] is an API set monitoring VM running on XEN. Secure hypervisor sHype [6] is basically an extension of secure OS based on MLS, RBAC and TE. However, XenAccess is mainly designed for memory dump and analysis of virtual machine. Particularly, XenAccess and sHype cannot monitor fine-grained resource access of Windows OS such as file R/W commit and registry access.

The proposed system is partly based on the concept of VMI (Virtual Machine Introspection) [7]. VMI is the ability of inspecting and understanding the events occurred inside virtual machine running on monitor. According to [7], hypervisor based monitoring leverages three properties of Virtual Machine Monitor (VMM), isolation , inspection and interposition. Revirt [8] is an extension of hypervisor for intrusion analysis of virtual machine. However, Revirt does not support real-time monitoring and analysis without suspending and snapshotting virtual machine.

Logging and debugging architecture of hypervisor has been proposed in [9] [10]. [9] points out that hypervisor technology enables centralized logging architecture as well as server consolidation. [11] leverages the utility of snapshot and reply of virtual machine for cyclic and deterministic re-execution. However, these systems are mainly designed only for debugging of memory analysis.

Volatility [10] is a tool for analyzing memory and retrieving information of virtual machine. Split device driver is inter-domain communication module of Xen. Also, these are not developed for real-time resource monitoring.

[12] and [13] propose the system for detecting event on the guest OS. These two papers solve semantic gap. In [12], Patagonix detects binary execution on the guest OS. It specifies what kind of binary executed is loaded and executed. In [13], the authors apply active monitor of virtual machine using active monitor. Lares, the proposed architecture, deploys security module monitoring memory operation for secure isolation of virtual machine. In the view point of active monitor, Lares is similar to our system. However, Lares is mainly designed for exploitation of memory and secure isolation. Also, its security module is architecture independent, developed on Windows OS and Xen while its authors do not describe file access monitoring, how deployable it is, and performance overhead. It is worth to point out that to detect many malicious software such as worm and trojan, monitoring file access is more important rather than real-time memory analysis. In this paper, we propose a more generic framework, protocol and implementation. The proposed generic protocol implementation enables us to monitor file access for both Xen and KVM, which is not achieved in the Lares system. Moreover, our file monitor driver can run on every kind of Windows OS: XP, Vista and 7. The proposed framework is architecture independent and allows reasonable overhead less than 5% for both guest and host OSes.

3 Hypervisor: Xen and KVM

In this paper, a monitoring system is implemented on two virtual machine monitors: Xen and KVM. The Virtual Machine Monitor (VMM), also called as hypervisor, is a thin layer of software between the physical hardware and the guest operating system. The rapid increase of CPU performance enables VMM to run several operating systems as virtual machine, multiplexing CPU, memory and I/O devices in reasonable processing time. Recent VMM is a successful implementation of micro kernels. Under the guest OS, VMM runs directly on the hardware of a machine, which means that VMM can provides the useful inspection and interposition of the guest OS.

Xen is a hypervisor booting and running under Linux kernel. Xen has its basic scheduler necessary for multiplexing virtual machines. KVM (Kernel Virtual Machine) is a Linux device driver based hypervisor, which work as a kernel module. KVM is more lightweight implementation but does not have its scheduler. Recently, Xen takes advantage on vCPU and I/O scheduling such as vcpu-pin, assigning and binding CPU for the guest and host VMs.

4 Communication protocol using virtual register

This paper proposes the two kinds of inter-domain communication system. Figure 1 shows type 1 which uses vCPU context for transferring access log to the host OS. The proposed system type 1 is implemented

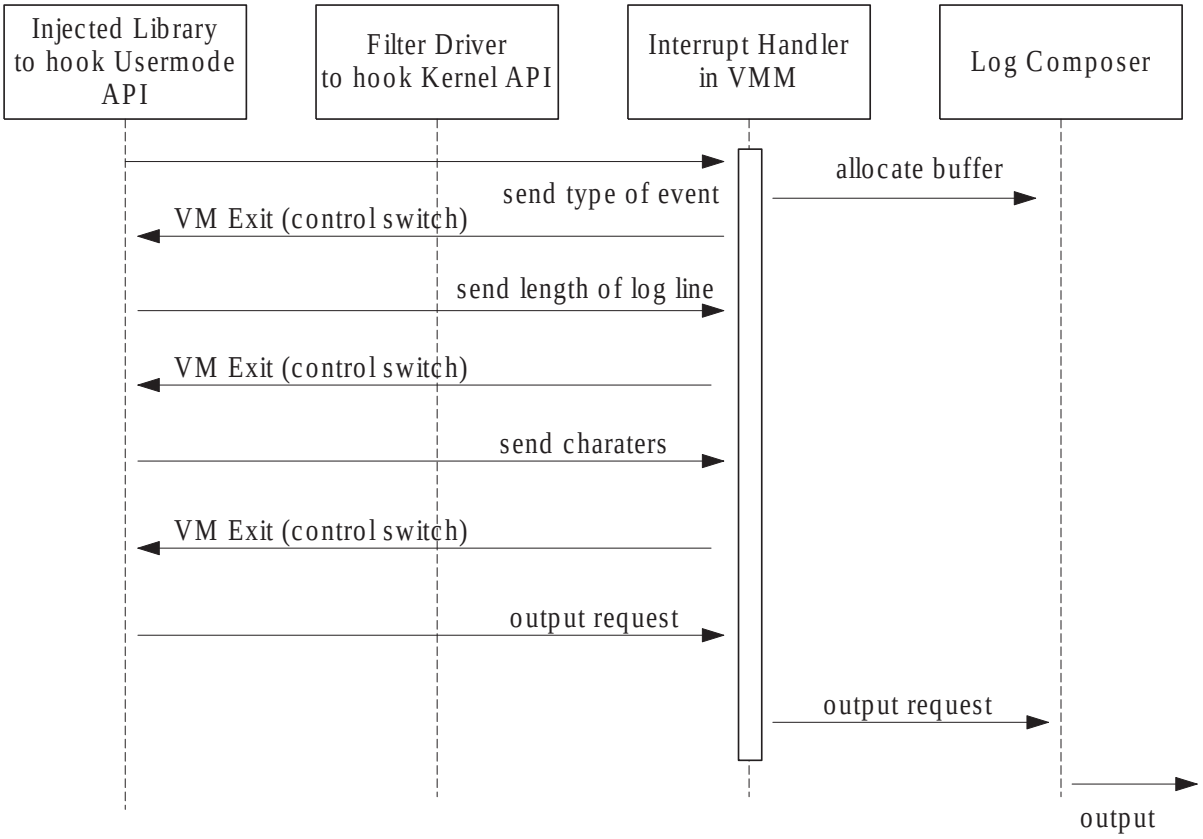


Figure 2: Sequence diagram for proposed inter-domain communication using degug register.

in three steps: [1]modification of Windows OS using DLL injection and filter driver, [2]modification of debug register handler of virtual machine monitor and [3]putting visualization tool on host OS. Figure 1 is a brief illustration of the proposed system. Our module extracts a sequence from Windows memory behavior and the sequence is transferred from the VMM module to the visualization tool on the host OS. In the following section, we also discuss modification of Windows OS. In this paper we propose the visualization of memory behavior of full-virtualized Windows OS using virtual machine introspection. In the proposed system, the memory behavior of Windows OS is visualized by an application of the concept of virtual machine introspection. The proposed system extracts a sequence of Windows memory behavior and transfers it to the host OS by modifying the module of virtual machine monitor. For simple implementation of the VMM side, the proposed system only monitors the changes of Debug Register(DR). When DR is changed by mov instruction, the DR handler in VMM is activated. At the same time, some values are moved to generic registers such as EAX, EBX, ECX and EDX, these values are also transferred. A line of log is transferred for one character, byte by byte inserted into a generic register. We add some header such as string length, type of events using other generic registers for each byte.

4.1 Modification of debug register handler

The x86 architecture has debug register DR0- DR6. Our system applies virtual machine introspection by monitoring these registers. Debug register is changed and accessed by register operation of MOV variants. When the CPU detects debug exception enabled, it sets low-order bits and then the debug

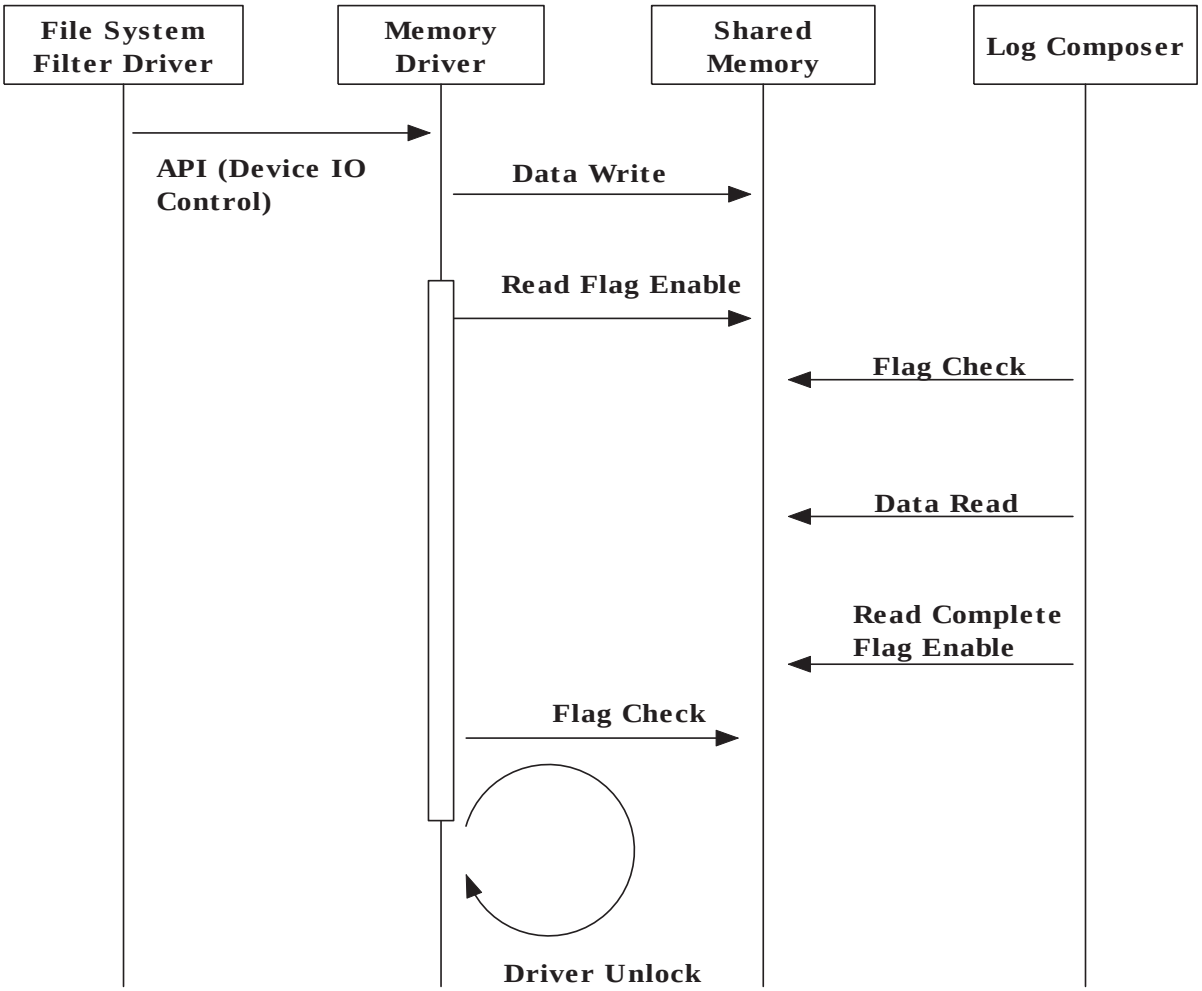


Figure 3: Sequence diagram for proposed inter-domain communication using ring buffer of shared memory.

register handler is activated. Among DR0-7, DR6 are never reset by the CPU. We also modify the debug register handler to receive information sent by our modules in Windows OS. the API is intercepted on the guest OS, the value of special register (DR/MSR) is changed. At that time, the context of virtualized CPU is stored in hypervisor stack. VMM can detect the access of the guest OS when domain context is switched because CPU context including the state of DR/MSR register is changed.

4.2 Transmission by virtualized register

Before activating the debug register handler in VMM, we divide each log line into byte-by-byte characters and translate these into character code. Character code and its header information is inserted into generic registers. After that, when the debug register is triggered, the proposed system can obtain the values of virtualized registers. Figure 2 shows the inter-domain protocol using VM enter/exit. There are four components in this protocol. Injected library and filter driver are running in the guest Windows OS. In VMM, interruption handler receives notification and message. Then, log composer output file access log to the host OS. These steps of transmission by virtualized register are shown as follows:

```
step 1: guest VM divides log string into
one character and store the value into
register.
step 2: guest VM changes debug register.
step 3: VM Exit is occurred.
step 4: Processor control is moved to
debug register handler on virtual
machine monitor.
step 5: host OS receives the value byte
by byte.
```

5 Communication protocol using shared memory

In this section we present a proposed communication system type 2 using shared memory. Figure 3 depicts our mechanisms of inter-domain connection. We achieve the fast inter-domain connection due to share a memory space between Dom0 and DomU. For example, Figure 5 shows a situation that two applications share each two memory spaces between DomU-1 and DomU-2.

This memory shared method enables that Dom0 user applications immediately can retrieve the data that is written by DomU. Each running PV-drivers on DomU-1 and DomU-2 allocate memory spaces to connect with Dom0 and write their signatures to the spaces. Dom0 can identify the shared memory space by scanning the signature.

As we will describe later, Dom0 handles shared memory space as ring buffer.

Figure 4 depicts the detail. DomU has a logging application as user-application on Dom0 to gather the data that is written by Dom0. On the other hand, the guest Windows OS on DomU has our PV-driver that provides some APIs to read/write data. Thus, any applications on DomU can send data to the logging application by calling this APIs. As stated above, this PV-driver has the shared memory space to connect with the logging application on Dom0.

We describe the mechanism by pursuing the flowing data between Dom0 and DomU. (A): First, PV-driver collects the data which is sent from applications through the driver APIs to ring-buffer. (B): Next, the logging application on Dom0 reads the ring-buffer. (C) Finally, the logging application saves the collected data to external files.

Figure 5 shows the flow chart of this protocol of which step is as follows:

```
Step 1: File System Filter Driver sends
log string to Memory driver
by calling APIs.
Step 2: Memory Driver writes log string
to its Shared Memory region
and sets a flag to indicate ready
state of data collection.
Step 3: Log Composer polls the flag.
When the flag is 1, Log
Composer collects data that was
written by Memory Driver.
Step 4: Log Composer sets a flag to
indicate the completion when
data reading is completed.
```

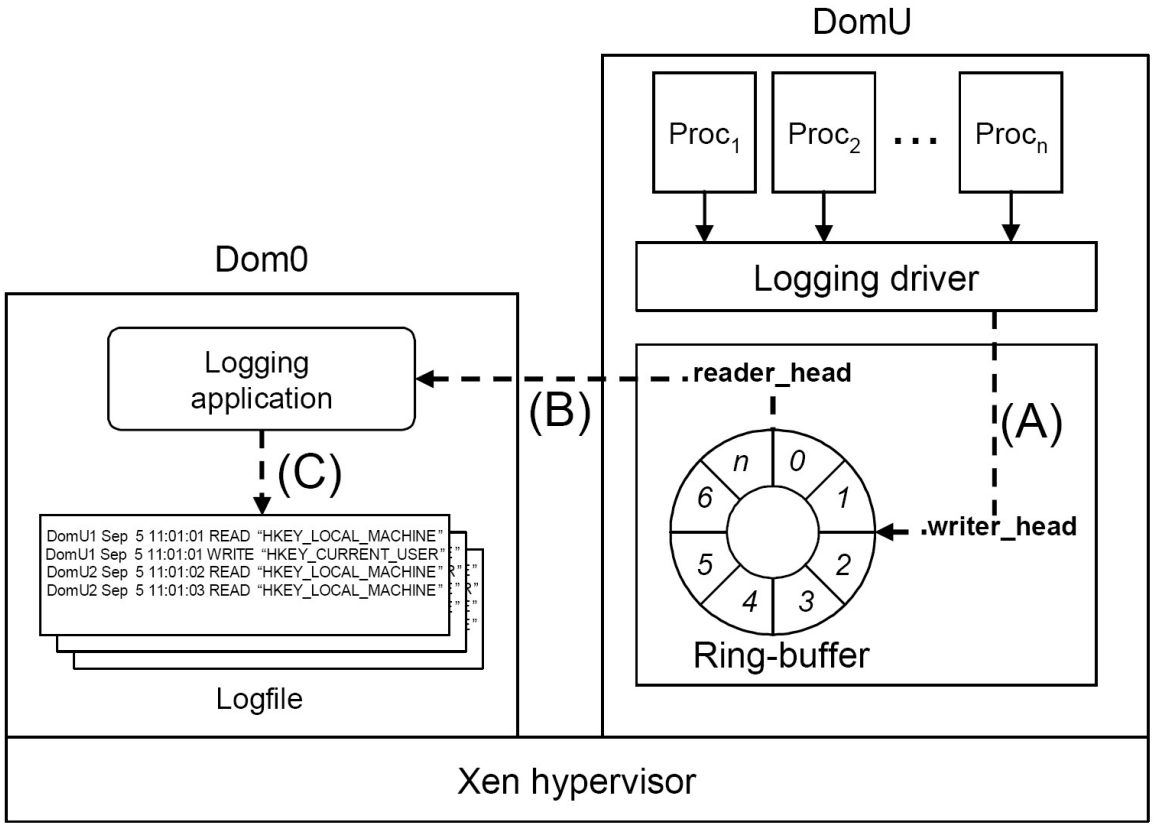


Figure 4: Logging architecture of virtual machine using shared memory

6 Windows OS modification

In the proposed system, we insert some debugging and monitoring modules to modify Windows user and kernel space. We apply two techniques: DLL injection and filter driver.

6.1 DLL injection

DLL injection is the insertion of original library on userland. We apply DLL injection for inspecting illegal resource access of malicious process. DLL injection is debugging technology to hook API call of target process. Windows executable applies some functions from DLL such as kernel32. dll. Executable has import table to use the linked DLL. This table is called as import section. Among some techniques of DLL injection, modifying import table is useful because this technique is CPU-architecture independent. Figure 5 shows the modification of import table. Address of function A on left side is changed to the address of inserted function on right side. In code table, some original functions are appended to executable. Modified address is pointed to code of inserted function. By doing this, when the function A is invoked, the inserted function is executed. In the proposed system, the inserted function changes special registers (DR/MSR) to notify the events to VMM and control domain.

6.2 File manager of Windows OS

Filter driver is an intermediate driver which runs between kernel and device driver. By using filter driver, we can hook events on lower level compared with library insertion technique on user land. In detail, System call table is modified to insert additional routine for original native API. In the proposed system, filter driver is implemented and inserted for hooking events on file system. Filter manager is supporting intermediate driver supplied by Microsoft that simplifies the third party's development. By using filter manager, we can add or replace functionality provided by the target driver of request.

6.3 File access filter

Figure 6 shows the sequence diagram of file access filter. Write request is divided into two action: PRE and POST. PRE operation is done before disk access. POST write is invoked after disk I/O complete. Figure 7 shows the specification of filter drier function. Intermediate function is implemented as callback function with three arguments and returns. By using this callback function, we can intercept and obtain the structure below in calling pre-write function. As shown in Figure 6, we have implemented inter-domain communication module after write request (PRE) is called.

7 Performance Measurements

7.1 Utilization of guest Windows OS

In this section, we discuss performance measurement of the proposed system. The proposed system is evaluated on KVM with linux-kernel 2.6.31 and XEN with linux paravirt kernel 2.6.31. Linux and Xen are running on NTEL(R) Core2 Duo with P8400 with 2GB memory. Virtual machine is also assigned by 800MB memory. Virtual Windows is Vista with service pack 2. On the first phase, we evaluate the proposed system by decompressing zip file on virtual Windows OS. Second, we set up Apache web server on VM and access it by using Apache bench. We have measured CPU utilization, available memory (KB) and the number of page faults for these two cases.

Figures 8 and 9 compare the CPU utilization of the guest and host OSes in decompressing zip file on Windows OS. In Figure 8, KVM based system has better performance with CPU idle time around 30 to 50%. On the other hand, XEN based system has CPU utilization with around 15%. Compared with native VM ENTER, invoking Hypercalls takes much more system resources. Concerning CPU time of the host OS, XEN based system has better performance. XEN based system takes about 5-10% idle time utilization while KVM based system takes about 30 to 15% CPU utilization. KVM has no interface to output system logs of VMM, so KVM based system needs to log to file system. On the other side, XEN based system has log output interface that is xm dmesg, which makes it more lightweight to run. Figures 10 and 11 show the available memory size in running the proposed system of KVM and XEN. In the guest OS, KVM has better performance. As is the case of CPU utilization, invoking hypercall takes more cost compared with transfer instruction of mov to debug register. Figure 11 shows that in the host and guest OSes, memory usage is not disturbed by the proposed system both of KVM and XEN based.

We have also measured resource utilization of virtual machine with Apache bench. Figure 11 shows the CPU utilization of the guest and host OSes with Apache web server receiving requests. In three items: processor idle time, available guest memory and page fault, KVM based module has better performance in the view of total utilization and the stability. In Figure 11, the CPU utilization is almost the same value (by 0.8 %). utilization of XEN based module is less stable than KVM. About memory utilization, KVM keeps more available memory then XEN by about 34% while the number of page faults is near the same by about 4% greater in XEN. In the case of Apache bench, CPU utilization has not been changed

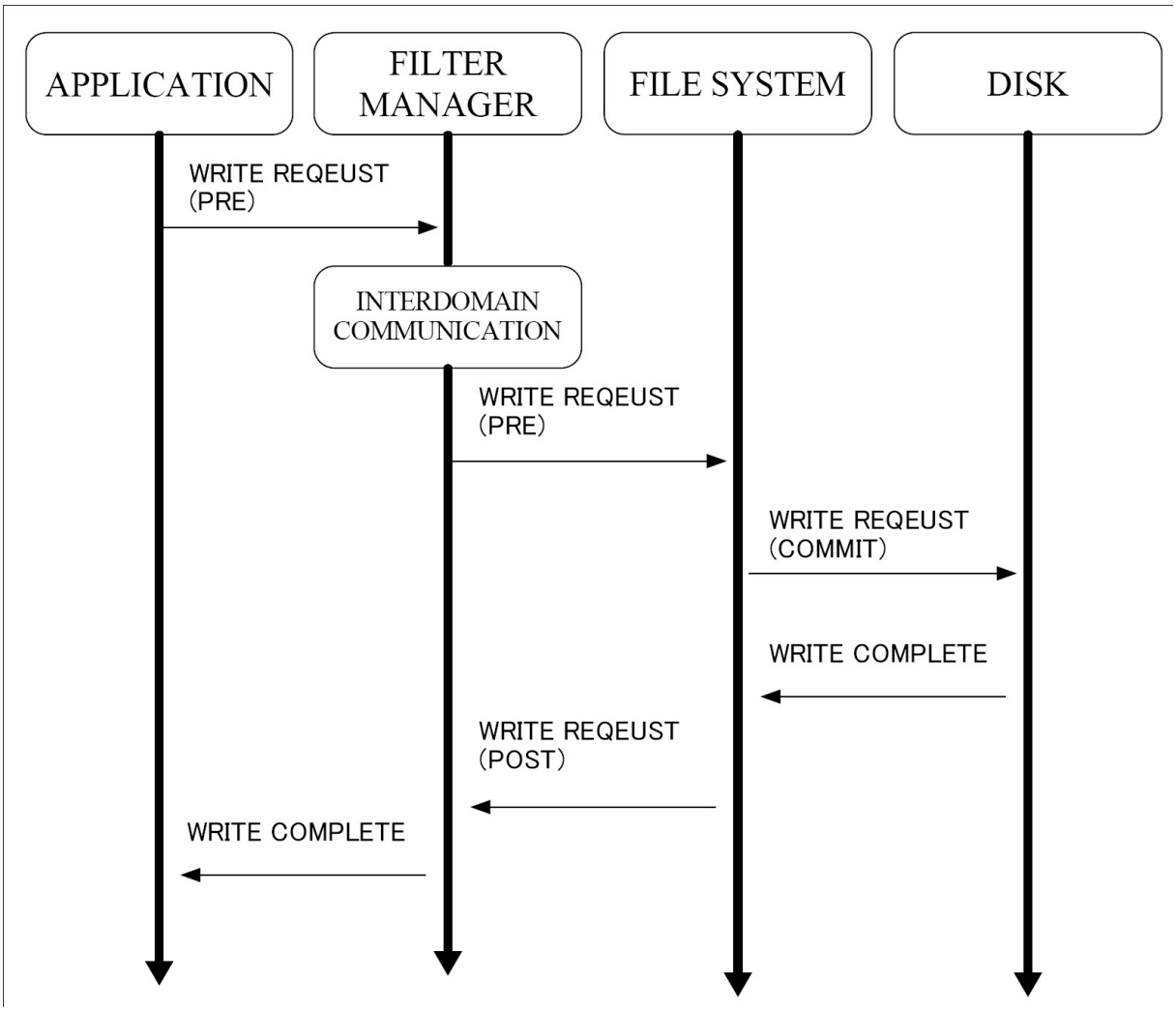


Figure 5: Sequence diagram of Windows filter driver. inter-domain communication module works between application and file system.

compared with the case of unzipping files. But memory usage is heavier with average with utilization of 600,000,000 of KVM and 650,000,000 of XEN. Invoking hypercall occurred by Apache request has more utilization than VM enter/exit of KVM.

7.2 Utilization of host OS

While KVM based module has better performance on the guest OS, about the utilization of the host OS (which is called Dom0 in XEN), XEN based system outperforms KVM. We have measured resource utilization of processor idle time and free memory of the host OS (Dom0) by using vmstat per one second. Figures 14 and 15 show the processor idle time when decompressing files and receiving massive http request. In the case of decompressing files, XEN based system has better performance by about 21%. Memory utilization is almost the same by About 1% difference between these. In the case of Apache bench, XEN based system has much less utilization by about 48% CPU idle time. Memory usage has slight difference.

type		FLT_PREOP_CALLBACK_STATUS
function		Obtains file information structure Control file access Returns value for file access
arg	**1**	PFLT_CALLBACK_DATA Data
	2	PCFLT_RELATED_OBJECTS FltObjects
	3	PVOID *CompletionContext
return	**1**	FLT_PREOP_SUCCESS_WITH_CALLBACK (succeed)
	2	FLT_PREOP_SUCCESS_NO_CALLBACK (succeed)
	3	FLT_PREOP_COMPLETE (succeed/fail)
failure		Return: FLT_PREOP_COMPLETE Set Data->IoStatus.Status error

Figure 6: Data structure of callbacks of Windows filter driver.

7.3 Utilization of file access monitor

We have compared the cases of the utilization of file access filter module with on and off. Figures 17 and 18 show the processor idle time of unzipping file. Also, Figures 19 and 20 show the CPU utilization in receiving HTTP request. We should point about that in Figure 18, XEN has better performance in keeping about four times CPU ilde time than KVM. In the case of massive file access such as unzipping files, XEN based system outperforms KVM based system on processor time.

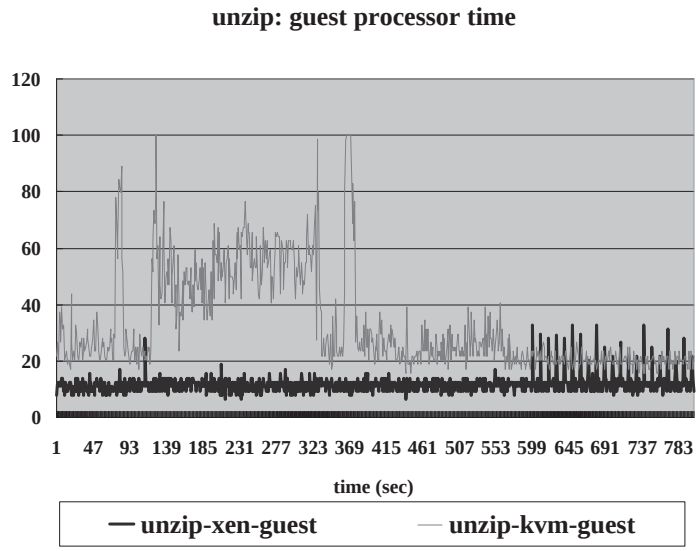


Figure 7: Processor idle time of guest OS in decompressing files.

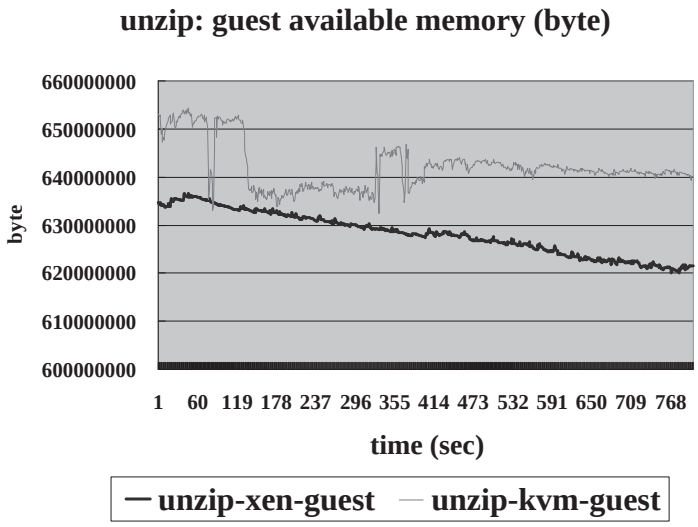


Figure 8: Guest available memory of guest OS in decompressing files.

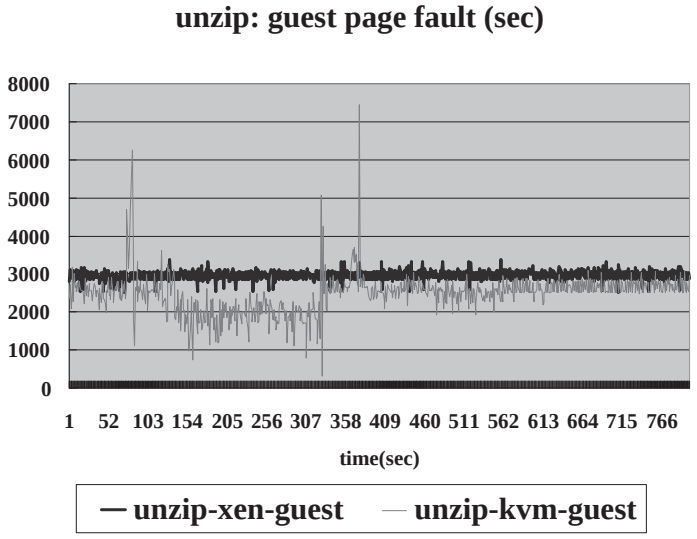


Figure 9: The number of page fault occurred in guest OS in decompressing files.

8 Discussion

For the guest OS, KVM based system is more lightweight about both CPU utilization and available memory than XEN based system. This is partly because that invoking and handling hypercall of XEN is more expensive than VM enter/exit of KVM. We can infer the reason from the number of guest page fault of which Xen based system is occurred more frequently than KVM based system as shown in Figures 10 and 13. In general, our inter-domain protocol implementation performs well with reasonable overhead less than 5% in both host and guest Operating Systems. However, performance is rapidly decreased in

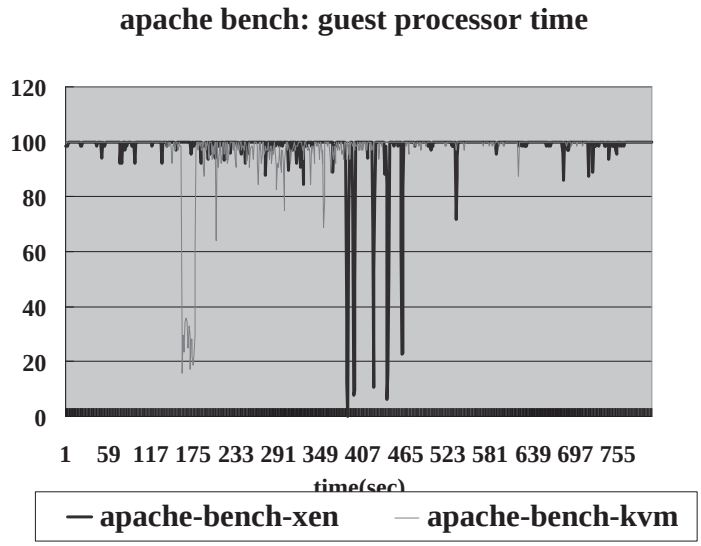


Figure 10: Processor idle time in guest of receiving HTTP request of Apache bench.

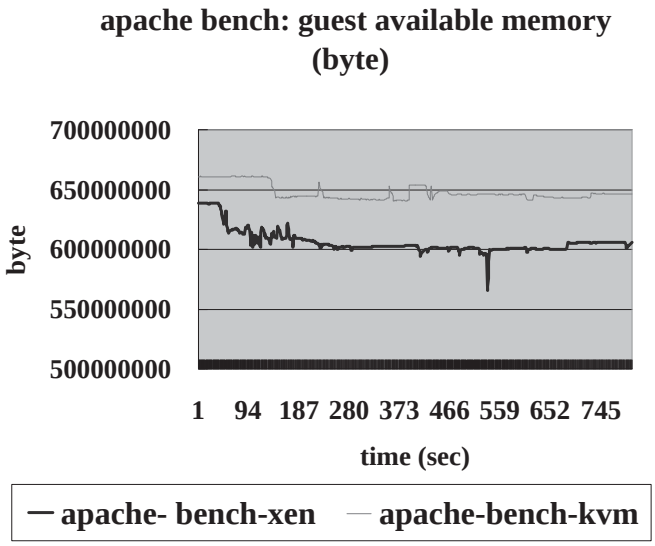


Figure 11: Available memory in guest of receiving HTTP request of Apache bench.

Xen based system coping with decompressing files as shown in Figure 19. We infer the overhead is caused by the cost of virtual I/O processing of Xen. About I/O scheduler, Xen based system is more expensive than KVM based system which leverages more improved Linux scheduler. From the results using Apache bench, our protocol implementation performs well for both XEN and KVM based system.

9 Conclusions

Virtual machine monitoring has been well researched. There still two remains: real-time resource access monitoring and active monitor. As virtual machine monitor running on hypervisor becomes pervasive

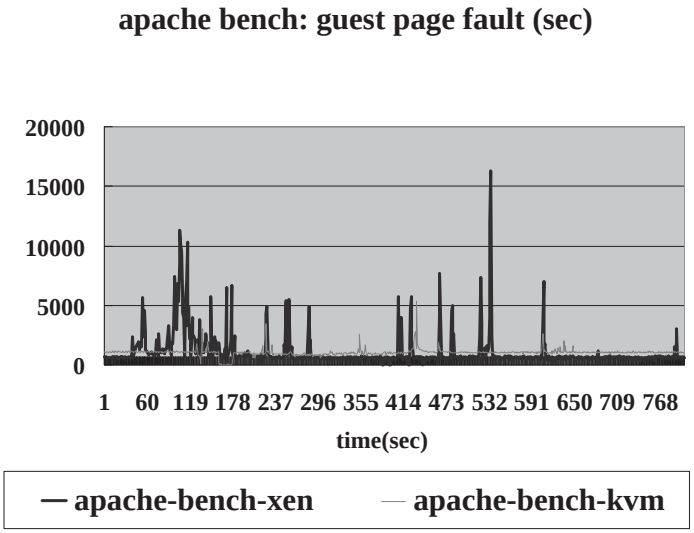


Figure 12: The number of page fault occurred in guest OS processing HTTP request of Apache bench.

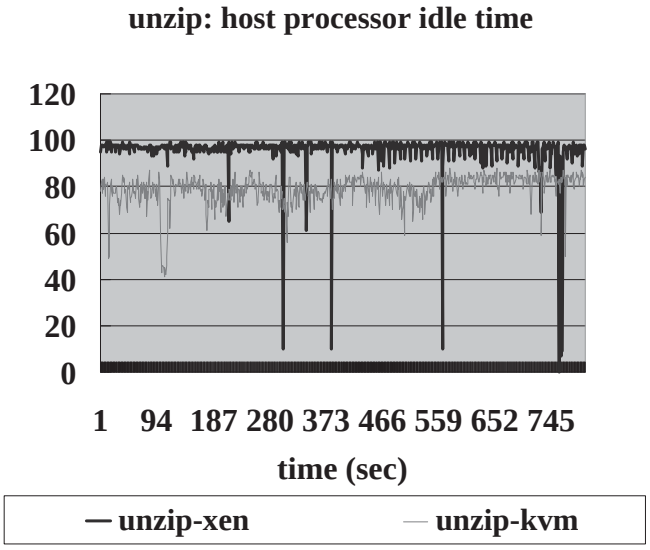


Figure 13: Processor idle time in host OS in decompressing files.

and ubiquitous, a generic inter-domain communication protocol and implementation, which can be applied for many kind of systems, is useful and actually necessary. In this paper, we propose an inter-domain communication protocol for real-time monitoring for detecting malicious file access on virtual machine. The proposed system can monitor resource access in real-time to detect malicious files access of virtual machine. Previous researches have paid much attention to memory analysis and architecture independent. Our proposal is a generic inter-VM communication protocol, which can be implemented on Windows XP, Vista and 7, and running on both the XEN and KVM hypervisors. We have implemented an inter-domain notification system using the kernel module of Windows OS and the virtual machine monitor in two ways: using vCPU context and ring buffer on shared memory between virtual machine

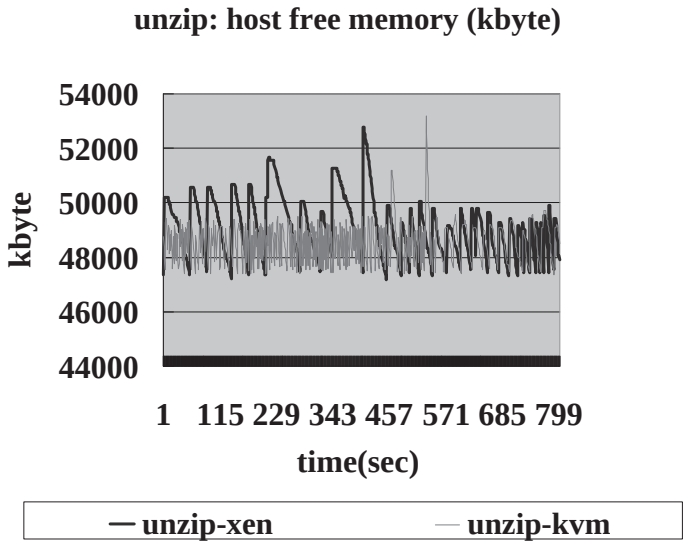


Figure 14: Free memory of host OS in decompressing files.

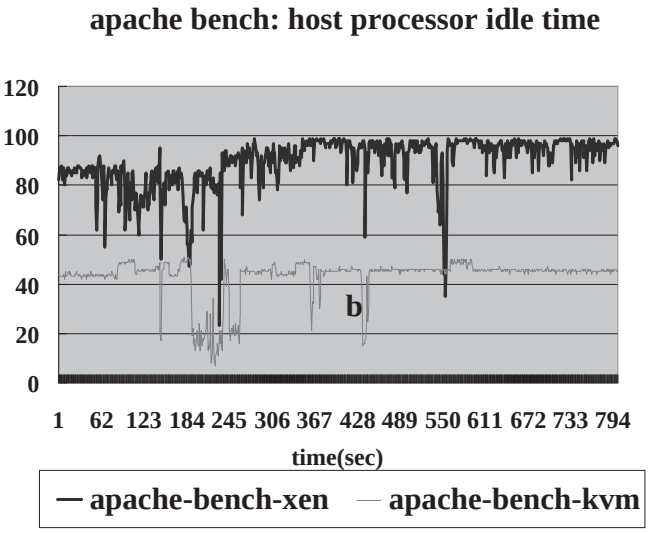


Figure 15: Processor idle time in host OS processing processing HTTP request of Apache bench.

and hypervisor. In the proposed system, we can monitor file access of the guest Windows OS on real-time, without suspending it. We have measured the resource utilization of these two systems in the case of decompressing files and receiving HTTP requests of Apache bench. On the guest OS, the KVM based system outperforms the processor idle time by about 30-50% in decompressing file and the memory usage by about 35% in receiving HTTP requests. It has been shown that with the proposed monitoring system which is activated, we can achieve lightweight resource utilization compared with no filtering case. We have also discussed the difference of the performance overhead according to the methodology and architecture of KVM and XEN. We can conclude it is possible to run real-time file access monitoring system without suspending virtual machine and with reasonable resource usage of less than 5%

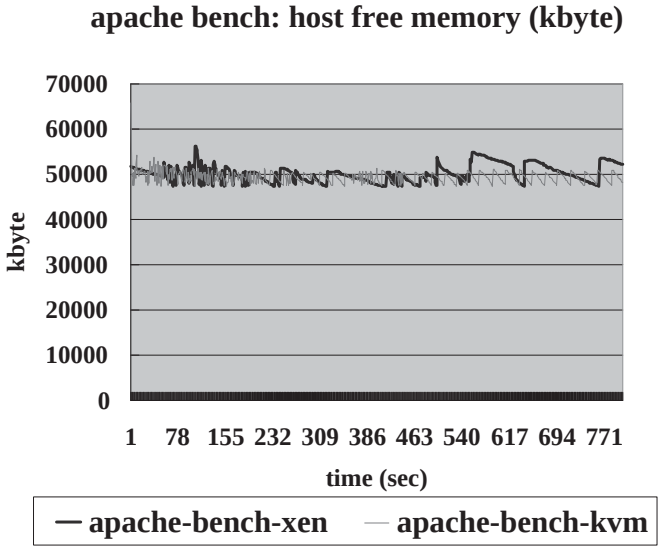


Figure 16: Free memory of host OS in receiving HTTP request of Apache bench.

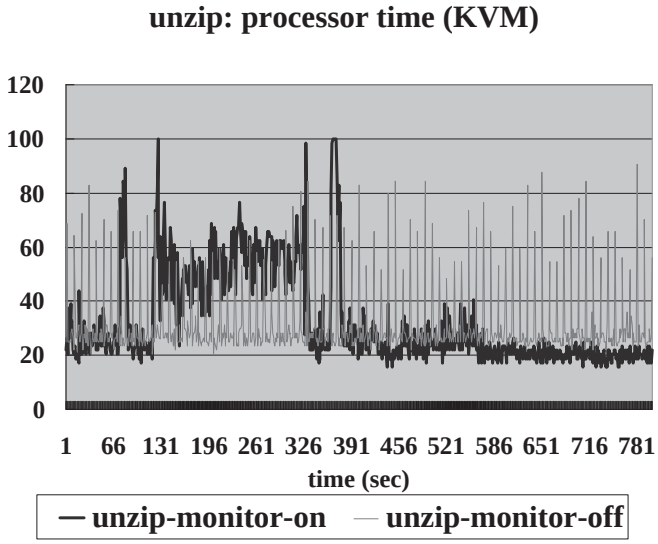


Figure 17: Comparison of processor idle time of guest OS of based system KVM in decompressing files

overhead. Moreover, our system can detect malicious file access.

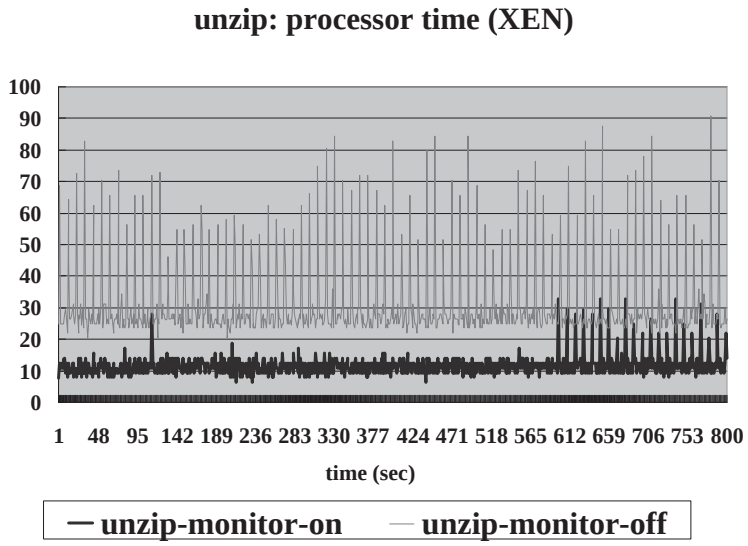


Figure 18: Comparison of processor idle time of guest OS of based system Xen in decompressing files

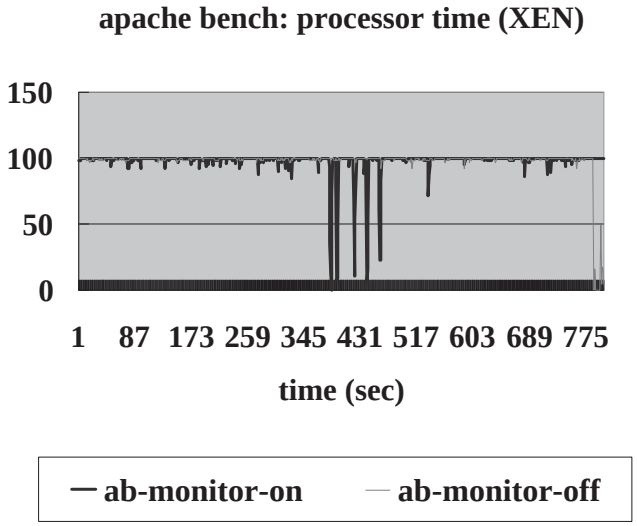


Figure 19: Comparison of processor idle time of guest OS of XEN in receiving reqeust of Apache bench

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, R. N. Alex Ho, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), New York, USA*. ACM, October 2003, pp. 164–177.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proc. of the 2007 Linux Symposium, Ottawa, Canada*, vol. 1, June 2007, pp. 225–230.

[3] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. of the 5th symposium on Operating systems design and implementation (OSDI'02), Boston, Massachusetts, USA*. ACM, December 2002, pp. 181–194.

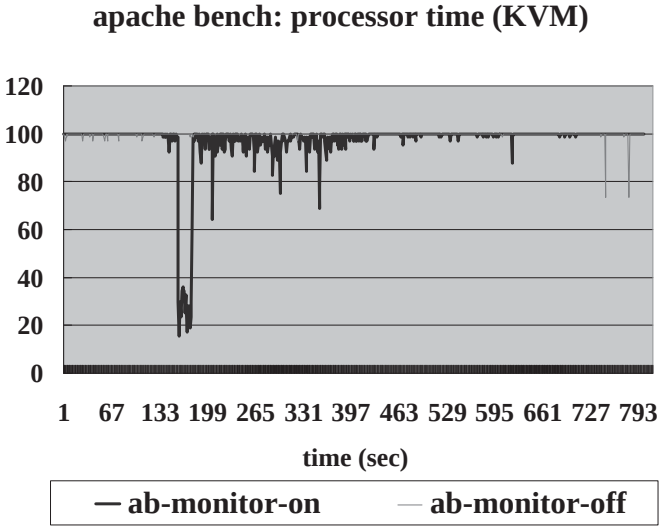


Figure 20: Comparison of processor idle time of guest OS of KVM in receiving reqeust of Apache bench

[4] M. Corporation, "Windows XP mode," http://www.microsoft.com/windows/virtual-pc/download.aspx.

[5] XenAccess, "Xen Access," http://doc.xenaccess.org/.

[6] IBM, "sHype," http://www.research.ibm.com/ssd_shype/.

[7] T. Garfinkel and M. Rosenblu, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. of 10th Annual Network and Distributed System Security Symposium (NDSS'03), San Diego, California, USA*, February 2003, pp. 191–206.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay," in *Proc. of the 5th symposium on Operating systems design and implementation (OSDI'02), Boston, Massachusetts, USA*, December 2002, pp. 211–224.

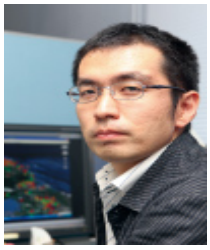
[9] A.-Q. Nguyen, R. Ando, and Y. Takefuji, "Centralized Security Policy Support for Virtual Machine," in *Proc. of the 21st Large Installation System Administration Conference (LISA'07), Dallas, Texas, USA*, November 2007, pp. 7–7.

[10] volatility, "The Volatility Framework: Volatile memory artifact extraction utility framework," https://www.volatilesystems.com/default/volatility.

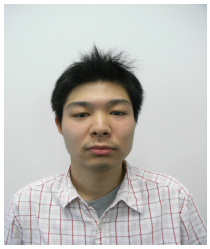
[11] S. King, G. Dunlap, and P. Cheng, "Debugging Operating Systems with Time-Traveling Virtual Machines," in *Proc. of the 2005 USENIX Annual Technical Conference (ATEC'05), Anaheim, California, USA*, 2005, pp. 1–1.

[12] L. Litty and H. A. Lagar-Cavilla, "Hypervisor Support for Identifying Covertly Executing Binaries," in *Proc. of the 17th conference on Security symposium (SS'08), San Jose, California, USA*, July-August 2008, pp. 243–258.

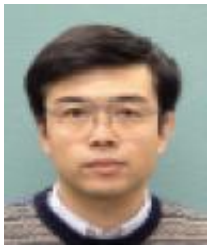
[13] B. P. et al, "An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. of 2008 IEEE Symposium on Security and Privacy (SP'08), Oakland, California, USA*. IEEE, May 2008, pp. 233–247.



Ruo Ando has received Ph.D. from Keio University in Japan. He is now senior security researcher of National Institute of Information and Communication Technology in Japan. Also, he has been working as Technical Official of Ministry of Internal Affairs and Communications since 2006. His research interests are cloud computing technologies and their security. He is the member of Trusted Computing Group JRF (Japan Regional Forum). He served as reviewer of Willey Journal of Security and Communications Networks and IEEE transactions of Information Forensics and Security.



Kazushi Takahashi is a doctor student of Graduate School of Information Science and Technology, the University of Tokyo. His research Area is virtualization technologies, especially OS-level abstraction and hypervisor.



Kuniyasu Suzaki was born in Tokyo, Japan and currently a senior researcher at RCIS (Research Center of Information Security), of AIST (National Institute of Advanced Industrial Science and Technology). He got BE and ME from Tokyo University of Agriculture and Technology, and received his Ph.D. in Computer Science from University of Tokyo. He joined ETL(Electrotechinical Laboratory) which has been reorganized into AIST. During ETL, he works for MITI (Ministry International Trade and Industry, which has been reorganized into METI, Japan) as a government staff. He stayed at Australian National University as Visiting Fellow (1997-1998) and at University of Illinois at Urbana-Champaign as Visiting Scientist (2001).