Investigating Timing Channel in IaaS

Rui Yang
Software Institute, Nanjing University
Jiangsu 210093, China
mg1432016@software.nju.edu.cn

Xiao Fu*
Software Institute, Nanjing University
Jiangsu 210093, China
fuxiao@nju.edu.cn
*the corresponding author

Xiaojiang Du
Department of Computer and Information Sciences,
Temple University, USA
xjdu@temple.edu

Bin Luo
Software Institute, Nanjing University
Jiangsu 210093, China
luobin@nju.edu.cn

ABSTRACT

In IaaS (such as Amazon EC2 and Microsoft Azure), several VM (virtual-machine) instances usually run in one physical machine so as to improve resource utilization. However this also caused more attack opportunities. A typical example is a cross-VM timing channel. Recent studies show that this kind of covert channel can successfully steal private information (e.g. private key) from the co-resident VM instances. It brought great challenges to the security of the cloud and has absorbed more and more attention in recent years. But to our knowledge, there is still little work on detecting and investigating such covert channel. Therefore, we propose a behavior-based method to automatically detect and investigate the timing channel. First, in order to record the behavior of this covert channel, a page-level memory monitoring method is designed. Second, an automatic identification algorithm is proposed based on some memory activity signatures. Finally, in order to confirm the result, the memory dump will be obtained and the binary code of the suspicious process will be analyzed. We have implemented a prototype on Xen, and the experimental results show that all of these kinds of attacks can be detected even under the disturbance from normal processes.

Keywords

cloud security, timing channel, Infrastructure as a Service, cloud forensics

1 INTRODUCTION

Timing channel is a leakage mechanism used to transfer and steal confidential information undermining the security of the operating system, database system and network [1]. This attack attempts to compromise a system by analyzing the time taken to execute some operations. Every logical operation in a computer takes time to execute, and the time can differ based on the input. With precise measurements of the time for each operation, an attacker can work backwards to the input [2]. In [3], Ristenpart et al proposed some timing channels in a cloud environment for the first time. They state that any physical machine resources multiplexed between the attacker and the target may form a potential leakage channel between the virtual machines [4]. These resources include network access, CPU branch predictors, CPU instruction cache, DRAM memory bus, CPU pipelines, scheduling of CPU cores, disk access, etc. In fact, cloud computing, especially IaaS (Infrastructure as a Service), is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources. In other words, resource sharing is very common in IaaS. So, timing channel will inevitably cause serious consequences.

In order to prevent this kind of covert channel, some protection mechanisms have been proposed. A typical example is sHype [5], which is a MAC-based (Mandatory Access Control) security extension to Xen hypervisor that allows the application of various security policies on VMs. In addition, some frameworks (such as HyperSentry [6], HyperSafe[7], Antfarm [8], HIMA [9] etc) were also proposed. They all focus on providing an integrity measurement to the hypervisor. However, these protection mechanisms may fail, because the timing channel is brought by sharing the resources. They observe behaviors of VM by monitoring hypercalls in a hypervisor and are looking for the regularity from system operation records. The system operations obtained are an abstracted behavior of the system layer, the information it contains is not complete. Timing channel uses time information by shared resources which cannot be reflected by the general system operation records.

In this paper, we designed and implemented a method, which can automatically identify and investigate timing channel in IaaS based on some behavior signatures. Usually, the presence of this kind of attack is difficult to find because it leaves no trace after the covert channel and all behaviors during the covert channel are legal and normal. This brought a great challenge to detection and forensics. Fortunately, after careful observation, we find that although single behavior is normal, these kinds of covert channels will show some regular patterns (called memory activity signatures) in long-term behaviors. These signatures will provide a good base to our automatic detection and forensic method.

Our method mainly includes three steps. First, in order to record the behavior of a timing channel, a page-level memory

This work is supported by the National Natural Science Foundation of China (61100198/F0207, 61100197/F0207).

MOBIMEDIA 2016, June 18-19, Xi'an, People's Republic of China

DOI 10.4108/eai.18-6-2016.2264107

monitoring method is designed. Second, an automatic identification algorithm is proposed based on some memory activity signatures. Finally, in order to confirm the result, the memory dump will be obtained and the binary code of the suspicious process will be analyzed by a static analysis method. A prototype system has been implemented on Xen. The design of this prototype can meet the following requirements: small modification to the protected systems, transparent to users, and acceptable performance impact. Some experiments have proved that our prototype can detect all of these kinds of attacks even under the disturbance from normal processes.

The rest of the paper is organized as follows. Section Ⅱ summarizes the related work and compares them with our method. Section Ⅲ introduces the design of our method in detail. Then, Section Ⅳ shows the experimental results. And Section Ⅴ concludes this work.

2 RELATED WORK

IaaS provides many kinds of shared resources that can be used by the timing channel, the most convenient resources to use are CPU, cache, memory, which makes the timing channel more effective.

Currently, the three typical categories of timing channel in IaaS are based on CPU cache, CPU load, and memory bus separately. [11, 12] used a cache-based covert channel to successfully decrypt the RSA and AES passwords. With a credit-scheduling algorithm on Xen, [13] implemented a CPU load-based covert channel between VMs. It can achieve better capacity and higher accuracy than the cache-based one. Ristenpart implemented a memory bus covert channel in the cloud [1], and then successfully improved its transmission rate up to 342 bps [10].

In order to prevent this kind of covert channel threat, several efforts have been made in recent years. A typical one is the hardware-based solution [10]. However, this solution usually has high costs and latency. Considering this, [14] designed HomeAlone to proactively detect the co-residence of unfriendly VMs so as to offer immediately deployable protection to IaaS. It detected the presence of a malicious VM by acting like a covert channel receiver and observing cache timing anomalies caused by another receiver's activities. But it cannot trace the evidence, our method can detect the existence of malicious behavior and collect related evidence of the timing channel, help build a more secure defense mechanisms.

Different with HomeAlone, XenPump [15] placed hooks into Xen hypervisor, monitored hypercalls and added some latencies to mitigate the threat of the timing channels, which will bring greater burden to the system. Our approach is proposed for the first time to observe the process behavior through low level memory activities, we use the EPT modification interface supplied by Xen directly in Dom0 to monitor memory activity, which cause small performance impact. We have not found other similar work so far.

3 SYSTEM DESIGN AND IMPLEMENTATION

3.1 System Overview

As Fig. 1 shows, our prototype mainly contains three modules, i.e. monitor, detector, evidence collector and verifier.

The goal of the monitoring module is recording the page-level memory activity of each process in real-time. The main work of the detector is identifying suspicious attack behaviors based on the signatures of timing channels from the records. Finally, because the result of the detector may contain some false alerts (some normal processes may behave like an attack), an evidence collection and verification mechanism is needed to confirm the result and provide more evidence.

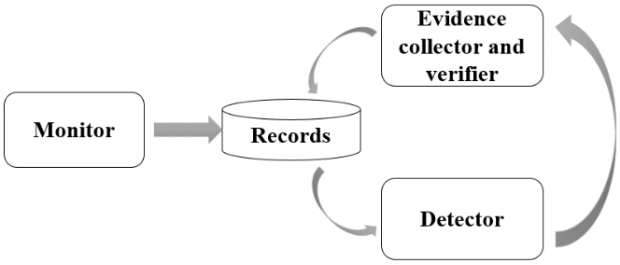

Fig. 1. System overview

Xen is one of the most popular VMMs used by IaaS providers. Moreover, it is an open source platform. So our method is currently implemented on Intel based Xen. We will extend this method to fit more platforms in the future.

3.2 Monitor

Monitor should meet some basic requirements, such as small modification to the protected systems, transparency to users, and acceptable performance impact. After careful study, we chose Intel VT EPT (Extended Page Table) [16] techniques to monitor guest VM's memory region. The monitoring information is delivered to Dom0 by event channels provided by Xen. No modifications of guest VMs are needed in the whole procedure. The guest VM is running normally and users will see nothing. What's more, EPT lowers the performance influence to guest VMs caused by our monitor module as much as possible. The LibVMI [17] API is used to realize the interaction between the Xen hypervisor and our monitor module. This makes the monitor module easier to implement.

EPT is a famous technique provided by Intel, which is used to reduce the difficulty of implementing memory virtualization and improve the virtualization performance. It uses a two-jump transformation (i.e. the logical memory address of the virtual machine to the physical memory address of the virtual machine to the physical memory address of the physical machine memory). Moreover, EPT provides a lot of new features to enhance memory virtualization, including isolation, rights protection, etc.

LibVMI is a C library with Python bindings that makes it easy to monitor the low-level details of a running virtual machine by viewing its memory, trapping on hardware events, and accessing the vCPU registers.

As Fig. 2 shows, an event channel between the target VM and our module is first initialized by the *vmi_init* function. Then, the *vmi_register_event* function will register memory events on the guest VM's memory region so as to monitor the activities of guest VM's memory in real-time. To trigger the event, the rights-protection mechanisms in EPT is adopted to set access permission on the target memory region. When there is a guest process which wants to access the memory pages, a violation event will be triggered and trapped in the hypervisor. Meanwhile, we will cancel the permission to let the guest process continue executing. The violation event will pass through the event channel in Xen hypervisor, then call the *mem_monitor_cb* function to handle it. Function *mem_monitor_cb* will record several crucial details such as page address, triggering time, page operation type, process

information. After handling the violation event, the *vmi_register_event* function will be invoked again to register the memory event so as to reset the access permission on that page and to wait for the next process access request.

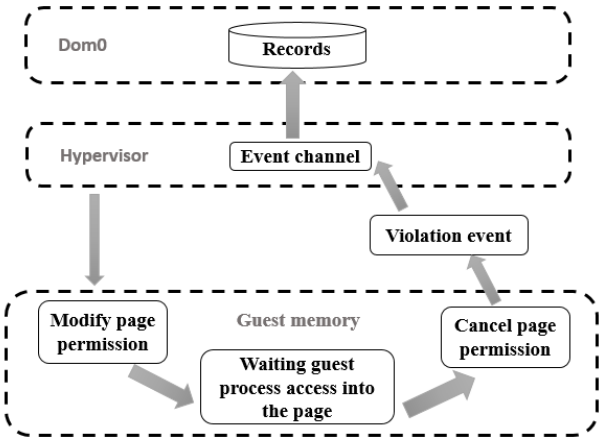

Fig. 2. Overview of Monitor

3.3 Detector

The Detector identifies the timing channels from large amounts of memory activity logs using the identification algorithm based on the memory signature, and filters the redundant logs. In this section, we summarized the memory signature of three popular kinds of timing channel attacks, and designed an identification algorithm based on these signatures.

The timing channel is mainly realized by encoding the information into time latency, and its basic agreement is the prime-probe protocol extensions [3, 12]. For example, the cache-based timing channel prime-probe protocol performs the following steps:

1) Prime: Read B at s-byte offsets in order to ensure it is cached.
2) Trigger: Busy-loop until the CPU's cycle counter jumps by a large value.
3) Probe: Measure the time it takes to again read B at s-byte offsets.

The transmission procedure of a timing channel is shown in Fig 3. The confidential information is transmitted between virtual machines which are not allowed to communicate directly. In the timing channel, the confidential information is encoded into the properties of different shared resources [15]. First, the sender encodes the information into binary bits. Then the sender changes the properties of the shared resources according to the bits. Finally, the receiver observers the changes and decodes the confidential information from these changes. The sender and receiver predetermine the parameters and repeat the cycle until all of the confidential information has been transmitted.

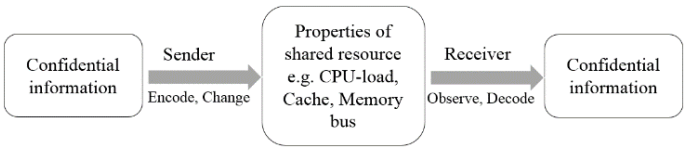

Fig. 3. Transmission Procedure of Timing Channel.

The three most popular kinds of timing channels are a CPU cache channel, load-based channel, and memory bus channel. Different shared resources have different information encoding positions. Cache-based timing channel is based on detecting the memory cache shared by two co-resident VMs, which can be approximated by the amount of time consumption for memory access latency. Load-based timing channel uses the CPU loads as the timing channel between VMs, which can be approximated by the amount of time consumption for certain computations. Memory bus timing channel uses the memory bus contention as the timing channel between VMs, which can be approximated by the amount of time consumption for memory access latency.

Before summarizing the memory signature of the timing channel, we need to demonstrate one important definition here.

DEFINITION. Intensive Memory Activity (IMA): Frequent memory access of a process in a short period of time, which show an obvious time interval with the next frequent memory access, are defined as Intensive Memory Activities.

TABLE 1. SIGNATURE OF THE TIMING CHANNEL

Type	Signature
Timing channel common signature	P_i: receiver process Q_i: sender process $\{j \mid j > 1, j \in N\}$ $P_i(\Delta t_j) = P_i(\Delta t_{j-1})$ $Q_i(\Delta t_j) \bmod \mathrm{MIN}(\Delta t) = 0$ $P_i(S_j) = P_i(S_{j-1})$ $Q_i(S_j) = Q_i(S_{j-1})$ $SIZE(P_i(S_j)) = SIZE(P_i(S_{j-1}))$ $SIZE(Q_i(S_j)) = SIZE(Q_i(S_{j-1}))$
Load-based	$P_i(t_j) = Q_i(t_j)$
Cache-based	$P_i(t_j) \neq Q_i(t_j)$ $\mathrm{TIME}\left(P_i(t_j)\right) < CacheLatencyThreshold$
Memory bus based	$P_i(t_j) \neq Q_i(t_j)$ $\mathrm{TIME}\left(P_i(t_j)\right) > CacheLatencyThreshold$

The memory activity signatures of the three timing channels as shown in Table 1. Due to the same prime-probe protocol they base on, the three types of channels have some characteristics in common. They will access the same memory region in fixed intervals and the sender's IMA intervals will be several times of the receiver's IMA intervals because of sending messages. The sender and the receiver will operate simultaneously in load-based channel and alternately in cache-based or memory bus channel. However, due to the memory access latency caused by the memory-bus lock being far longer than the cache's, we can simply distinguish the cache-based and the memory bus channel. In the following description of memory signatures, P is the set of VM sender processes and Q is the set of VM receiver processes. In other words,

$$P = \{P_1, P_2, \ldots\}, \mathrm{Dom}(P_i) \in VM_{receiver}$$

$$Q = \{Q_1, Q_2, \ldots\}, \mathrm{Dom}(Q_i) \in VM_{sender}$$

t_i denotes the start time of the i-th IMA. Δt_i denotes the interval of the i-th adjacent IMA. S_i denotes the set of the i-th IMA active pages. $SIZE(S_i)$ denotes the total size of the i-th IMA pages. $TIME(t_i)$ denotes the whole time of the i-th IMA. $P_i(t_i)$,$P_i(\Delta t_i)$ and $P_i(S_i)$ represent the start time of the i-th IMA of the process P_i, the interval of the i-th adjacent IMA of the process P_i, and the set of the i-th IMA active pages of the process P_i.

According to the signature of timing channel, an algorithm is designed to identify suspicious processes. First we look for the page that has been repeatedly accessed in line 3, and then the corresponding process is marked. After computing the interval of each IMA and the accessed pages of each IMA from line 9 to line 12, we compare them with the timing channel common signature from line 16 to 20. Finally, the timing channel type is determined according to the start time and memory access latency of the two attack processes in each VM IMA from line 24 to 29.

Algorithm 1 identify timing channel

```
1:  SuspiciousProcess = empty
2:  for i= 1 to recordSize do
3:    finding repeated page and identify intensive memory activity
4:    if exist repeated page then
5:      SuspiciousProcess.add(the process which accessed the repeated page)
6:    end if
7:  end for
8:  for each process in SuspiciousProcess do
9:    process.t.add(the start time of the first page of each intensive memory activity )
10:   process.dt.add(the interval of each adjacent intensive memory activity )
11:   process.S.add(each intensive memory activity pages)
12:   process.N = how many times the intensive memory activity
13: end for
14: for each process in VM sender and VM receiver do
15:   for j = 2 to process.N do
16:     if process.dt[j] mod MIN(process.dt[j] (2<j<process.N)) != 0 then
17:       SuspiciousProcess.remove(process)
18:     end if
19:     If process.S[j] != process.S[j-1] then
20:       SuspiciousProcess.remove(process)
21:     end if
22: end for
23: for suspicious process in VM sender and VM receiver do
24:   If process_sender.t[j] == process_receiver.t[j] (2<j<process_sender.N) then
25:     attack_type = load_based
26:   else if MAX(TIME(process_receiver.t[j] (2<j<process_receiver.N))) > cacheLatencyThreshold then
27:     attack_type = memoryBus_based
28:   else if MAX(TIME(process_receiver.t[j] (2<j<process_receiver.N))) < cacheLatencyThreshold then
29:     attack_type = cache_based
30:   end if
31: end for
```

3.4 Evidence Collector and Verifier

The Evidence Collector and Verifier module further confirms the presence of timing channel and collects related evidence. After a lot of experimental observation, we found some normal processes are marked as suspicious processes in the results of the detector module. For example, the clock process and backend process also access a fixed memory page area in fixed intervals. So they have a similar signature to timing channel. In order to avoid such misjudgment and increase the credibility of the investigation, we need to further check the suspicious process to prove the real existence of such an attack.

The Verifier is implemented as a volatility [18] plugin, which is a memory extraction utility framework. We find the memory address where the suspicious process through linux_proc_maps plugin, the linux_dump_map will help us copy out the data of the suspicious process.

After obtaining the binary code from the suspicious process, using objdump tool to disassemble the code, we then perform the static analysis based on the timing channel code signature. This step is to further confirm the presence of the timing channel to reduce the false positive, and also to obtain the communication protocol from the analysis of the communication code.

The memory dump and memory activity records of the attack will be kept as evidence of the attack when the attack is confirmed. Furthermore, these memory activity records can help us restore the transmission process of the attack and extract the transmission of information. The main steps of this module are shown in Fig. 4.

Fig. 4. Main steps of the Evidence Collector and Verifier

4 EXPERIMENTS

We implemented our prototype system on a desktop computer with Intel(R) Core(TM) i5-3330 3.00GHz CPU, 8G main memory, and 256KB L2 cache. The version of Xen hypervisor was 4.4.3. The version of the LibVMI was 0.10.1. We ran Ubuntu14.04 in Dom0 and two guest VMs, each of which was allocated 1024 MB of virtual memory.

In order to simulate timing channels: two DomUs set their vCPU to 1 in Dom0, both are pinned to the same physical CPU core, Xen enables default credit task scheduling algorithms, and two DomU credit values are set to (100, 0). Our forensic system is tested by three experiments, which are to identify CPU cache-based, load-based and memory-bus-channel attacks. We implemented the three kinds of timing channels based on the prime-probe protocol.

4.1 Simulation of Cache-based Channel

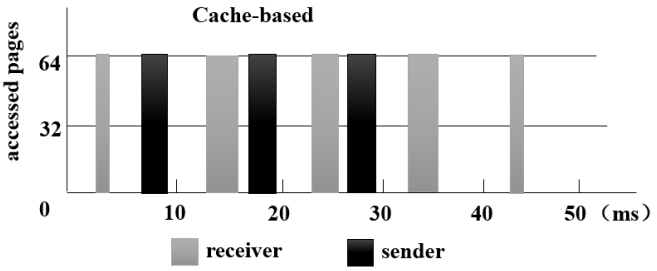

Fig. 5. Cache-based channel memory activity records

In the cache-based channel simulation, we focus on the L2 cache side channel. When the CPU accesses the memory, it first extracts the contents of memory to the cache during the calculation, if there is no cache of the content it will trigger a cache miss event, and then the cache will extract content from memory, thus brings latency to the entire computing. We applied for a 256 kb size of memory pages, just as the size of the L2 cache. To send a bit, the sender evicts the whole L2 cache by accessing the above applied

memory, and the receiver can decode the information by probing the memory access latency.

As Fig. 5 shows, the execution time of the sender and receiver is crossed, and the receiver memory access latency is obvious. We can get a message 01110 from this period of the memory access log.

4.2 Simulation of Load-based Channel

In the Load-based channel simulation, since CPUs or cores are often shared among different virtual machines and processes, a malicious process (e.g. spyware) on one virtual machine can secretly communicate with its peer on another virtual machine by changing the CPU load and making the peer recognize the change. We created a task *t* of which the execution time is 260ms; it is executed in the VM sender process and VM receiver process. The sender can send information to the receiver by changing CPU load according to a communication protocol, such as "bit 1 stands for being sent if sender is executing *t*, and bit 0 is for being sent otherwise."

As Fig. 6 shows, the task execution of the sender and receiver start at the same time, and the task latency is obviously observed. We can get a message 10110 from this period of the memory access log.

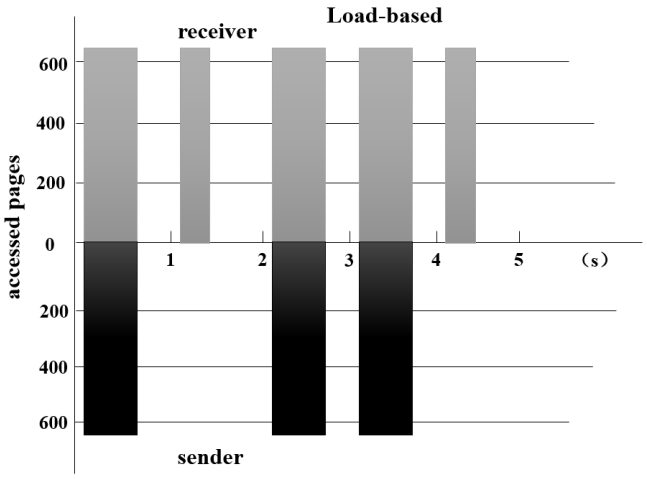

Fig. 6. load-based channel memory activity records

4.3 Simulation of Memory bus Channel

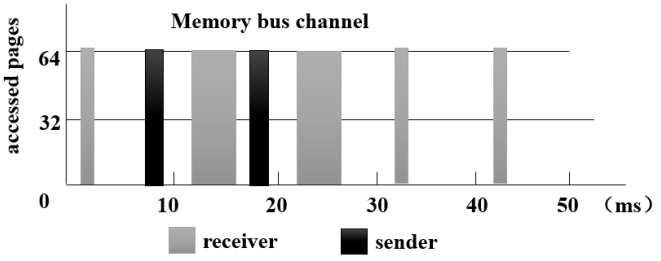

Fig. 7. memory bus channel memory activity records

In the memory bus channel simulation, the basic idea is the contention on the memory bus will result in a system-wide observable effect of increased memory access latency. We applied for the size of a L2 cache memory area. The sender uses mutex and semaphores to perform an atomic memory operation to cause the memory bus lock. Then the receiver can decode the information by probing the memory access latency.

As Fig. 7 shows. Like the cache-based channel, the receiver performance on memory access latency in the memory bus channel is longer. We can get a message 01100 from this period of the memory access log.

4.4 Detection and Confirmation of the Attacks

We tested each of the three types of timing channels 30 times in our experiments. The result that the attack is confirmed after it is identified by our experiments is shown in Table 2.

TABLE 2. EXPERIMENT RESULT

Channel	Test times	Identified times
Cache-based	30	30
Load-based	30	30
Memory bus	30	30

After the detector module identifies suspicious processes, there are possibly some normal processes in the result. So we use the Volatility to obtain the binary code of the suspicious processes based on the memory dump. Then the suspicious processes are checked by static analysis so as to confirm whether they are the timing channels. The experiments shows that this process can reduce the false positive rate to 0 successfully. Moreover, if we increase the repetition rate (i.e. the frequency of accessing the same memory page in a second) threshold of the identification algorithm in the detector module, as shown in Fig. 8, the number of misjudgment processes continues to decline, but at the same time the completeness of detection will also decline.

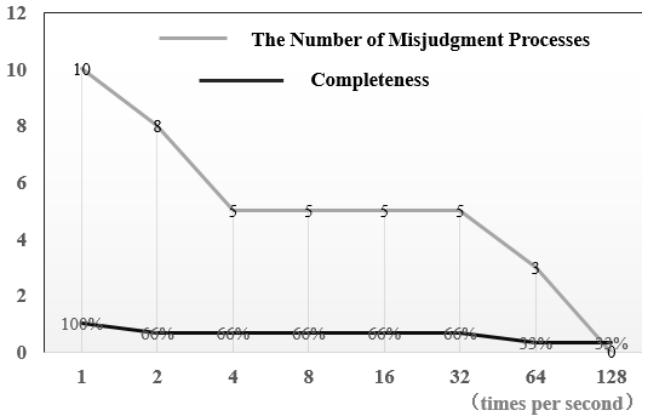

Fig. 8. The false positive rate with different repetition rate threshold

4.5 Performance

Our prototype system is running under a typical environment in our experiments. If more user processes are added, the memory activities will be highly frequent. That will cause a certain overload to real-time monitoring. In order to evaluate the impact on performance caused by our system, we compared the memory access latency of three types of simulated attacks under the condition of running our system or not.

As is shown in Table 3, we found that the time delay caused by the monitoring system is relatively fixed; each memory event caused the delay for about 200ns. In our simulation experiment, about 10000 memory activity events per second will cause around 2ms' latency on average. This is acceptable for the users and does not affect the stability of the guest VM.

TABLE 3.PERFORMANCE IMPACT

Channel	Performance impact
Cache-based	decreasing 10%
Load-based	decreasing 0.01%
Memory bus	decreasing 4%

4.6 Discussion and Future Work

As for the performance, the system delay is within an acceptable range, but at the peak of the operation of the system there is still a big impact. Although the performance can get close to the real physical machine with the Intel VT technology and the EPT, the whole procedure in Xen transmission channel is still too long and the entire delay occurs mainly in the process of transferring and handling events. In the future, we intend to build a monitor module directly from the hypervisor, leave out redundant steps, and separate the event trigger and event handling into different threads, thus reducing the impact on the system performance.

In the paper, we have tested three common timing channels which are implemented according to the traditional prime-probe protocol. However, the upgraded versions of these attacks are not tested, and neither are other types of attacks on timing channels such as shared memory, hard drive, etc. We will summarize more memory signatures of these timing channel to further improve our identification algorithm. We believe the basic idea of our method is fit for many kinds of virtual machine systems. We will further implement the prototype on other virtualization platforms such as VMware VBOX, etc.

5 CONCLUSION

In the paper, we investigated timing channel in IaaS, summarized the memory signatures of these covert channels, and then proposed a method to identify and investigate timing channels based on the signature. We have implemented our prototype system in Xen. Our experiment results show that the prototype can successfully identify these attacks. Moreover, our prototype can meet the following requirements: small modification to the protected systems, transparent to users, and acceptable performance impact.

6 REFERENCES

[1] NCSC, "Trusted computer system evaluation criteria (orange book)," 1985.

[2] T. V. Vleck. Timing channels. Poster session, IEEE TCSP conference, 1990.

[3] Ristenpart T, Tromer E, Shacham H, et al. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds[J]. Ccs Conference, 2009:199-212.

[4] Wu J, Ding L, Wang Y, et al. Identification and Evaluation of Sharing Memory Covert Timing Channel in Xen Virtual Machines[C]// 2012 IEEE Fifth International Conference on Cloud Computing. IEEE, 2011:283-291.

[5] R. Sailer, T. Jaeger, E. Valdez, R. C´aceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," in ACSAC. IEEE Computer Society, 2005, pp. 276–285.

[6] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in ACM Conference on Computer and Communications Security, 2010, pp. 38–49.

[7] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in IEEE Symposium on Security and Privacy, 2010, pp. 380–395.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in USENIX Annual Technical Conference, General Track, 2006, pp. 1–14.

[9] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in ACSAC, 2009, pp. 461–470.

[10] Wu Z, Xu Z, Wang H. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud[C]//USENIX Security symposium. 2012: 159-173.

[11] Bernstein D J. Cache-timing attacks on AES[J]. 2005.

[12] O. Acii¸cmez, W. Schindler, and ¸C. K. Ko¸c. Cache based remote timing attack on the AES. In Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, pages 271–286, February 2007.

[13] Okamura, K., Oyama, Y.: Load-based covert channels between Xen virtual machines. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 173–180. ACM, New York (2010).

[14] Zhang Y, Juels A, Oprea A, et al. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis[C]// 2011 IEEE Symposium on Security and Privacy. IEEE Computer Society, 2011:313-328.

[15] Wu J, Ding L, Lin Y, et al. Xenpump: A new method to mitigate timing channel in cloud computing[C]//Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE, 2012: 678-685.

[16] Intel: Hardware-Assisted Virtualization Technoloty. URL: http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html (2014)

[17] Payne B, De Carbone M, Lee W. Secure and Flexible Monitoring of Virtual Machines[J]. Computer Security Applications Conference 2007 Acsac 2007 Twenty Third Annual, 2007.

[18] The Volatility Framework, https://code.google.com/p/Volatility/; 2014.