Controlling the Speed of Virtual Time for Malware Deactivation

Keisuke Okamura Yoshihiro Oyama
The University of Electro-Communications

Abstract

We propose a mostly OS-independent, VMM-based method that deactivates malware at the granularity of a process. Specifically, the method slows malware processes extremely by shortening the timer interrupt intervals and modifying the system time value: the amount of time that elapses from the boot. We implemented a VMM based on the method, named *HyperSlow*, and confirmed that it can slow a particular process considerably.

1 Introduction

Infrastructure as a Service (IaaS) is an attractive hosting service of cloud computing. Users of an IaaS service manage virtual machines (VMs). An IaaS provider manages virtual machine monitors (VMMs) and physical machines. Each VM running on a VMM is managed by the service user to which the VM is assigned. An OS running in a VM is called a guest OS. The root privilege of the guest OS is held by the service user, not by the IaaS provider.

Here we consider the case in which malware compromises a certain guest OS and in which the guest OS administrator is unaware of the compromise. The malware might consume large amounts of computing resources in the VM, thereby slowing other VMs running on the same VMM. Alternatively, the malware might perform inappropriate communication with an external machine such as sending spam e-mails. The VMM administrator would hope to take action against the malware to maintain the quality of the service and the trust of the provider, even if the administrator of the guest OS does not explicitly allow it.

In this case, existing systems can take only coarse-grain actions such as stopping the whole VM, assigning an extremely low priority to the VM, or dropping all communication packets sent from the VM. The granularity of all actions is a VM. Therefore, when at least one action is taken, other good services managed by the VM also stop.

To address the problem, a fine-grained malware prevention method that affects only a particular process is required. The method should not assume a particular OS because OSes of various kinds will run in a VM.

This paper describes a malware prevention method by which a VMM mostly deactivates the execution of a specific process running in a VM. The method, which remarkably reduces the execution speed of an arbitrary process running in a VM, achieves that reduction by shortening the intervals between virtual timer interrupts and by emulating the pace of virtual time. This method assumes only a few things about the implementation of a guest OS. It depends little on the implementation. Therefore, it is applicable to OSes and OS versions of various kinds.

APSys '12, July 23-24, 2012, Seoul, S. Korea

We implemented a system that incorporates the method by extending Xen 4.0.0. The system is named HyperSlow. We implemented HyperSlow for Linux and the x86 architecture. We assume that an IaaS provider manages Domain 0 (dom0) and the hypervisor, although individual IaaS users (customers) manage their respective domains U (domU). Modifying a guest OS kernel or running a daemon in a guest OS is not necessary.

Indeed, although slowing a malware process is a mild countermeasure, it is an effective measure that the VMM can take when killing or stopping a process depends on the implementation of the guest OS *and* when the VMM must minimize the effect on benign processes. It should also be considered that no malware detection method can achieve 100% accuracy. Therefore, the malware detection might be a false positive. We consider that remarkably high accuracy of detection is necessary to execute strong countermeasures without contacting the guest OS user. We expect that an IaaS provider uses the method as the first mild response to minimize damage. While the method is working, the IaaS provider contacts the guest OS user and urges the user to take some action such as killing the malware process.

2 CPU Management in Xen

The Xen hypervisor provides virtual CPUs (VCPUs) to VMs by virtualizing the physical CPUs. A guest OS recognizes VCPUs of its VM and schedules processes on the VCPUs.

The *domain scheduler* in the hypervisor maps VCPUs to physical CPUs (PCPUs) dynamically. The domain scheduler distributes CPU time to each domain, whereas the process scheduler in each guest OS kernel distributes CPU time to each process.

Modern OSes such as Linux and Windows schedule processes onto CPUs based on the number of timer interrupts and/or the variables which indicate the amount of elapsed time. The variables are often calculated using the values of hardware clocks such as the time stamp counter (TSC). The Xen hypervisor delivers virtual timer interrupts to VMs at a certain interval and provides the value of *system time* (the amount of time that has elapsed since the boot) to each domain. In the Xen source code, the system time value is maintained by a member in a structure representing the time information of a VCPU (`system_time` in `struct vcpu_time_info`).

Currently, the standard scheduler in Xen is the credit scheduler, which works according to two key parameters associated with each domain: *weight* and *cap*. They are specified by the Xen administrator. The amount of CPU time distributed to a domain, called *credit*, is determined based on the respective weights of all domains. The cap of a domain specifies the maximum amount of PCPU time which the domain receives. The scheduler uses a fixed time slice in PCPU scheduling. Each time the slice elapses, the scheduler chooses a VCPU to which PCPU is assigned during the next slice. Domains that have not consumed all credits are scheduled in a round-robin manner. Even if other domains are idle, the hypervisor does not provide PCPU time to a domain in which the consumption of CPU time has reached its cap.

3 Proposed Method

3.1 Overview

We consider a case in which a malware process is running in a domU (Fig. 1). We designate such a domain as a *maldomain*. Domains without malware, which we call *benign domains*, are also running on the same VMM. Normal application processes are running in the maldomain. The IaaS provider manages the dom0 and hypervisor, whereas users manage their own domU.

The first class of malware we are concerned with includes malware that monopolizes

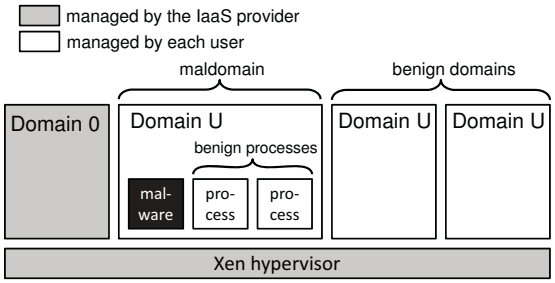

Figure 1: Imaginary environment.

computing resource and executes a Denial-of-Service (DoS) attack against other processes or domains. Victim processes or domains under this attack experience a slowdown of programs because of the decrease in computing resource assigned to them. The computing resource includes CPU time, physical memory, and network bandwidth. The second class of malware examined here includes malware that executes network-based attacks (e.g., sending numerous spam e-mails and sending a flood of requests to Web servers).

The proposed method disturbs malware by accelerating the speed of virtual time only while a PCPU is assigned to a malware process. It changes the timing of virtual timer interrupts and the value of system time. It depends little on the implementation of a guest OS because it varies the speed of virtual hardware only. Unfortunately, the method cannot stop the execution of malware completely, in principle.

Generally, recognizing a process from the hypervisor layer is not straightforward because the representation of a process depends on the OS implementation. However, a previous paper [5] described a technique for recognizing a process from a value in the page table register (CR3 in x86 processors). We also use that technique.

For this study, we do not assume a particular malware detection method because detection is mostly orthogonal to prevention. Several reports in the relevant literature have proposed security systems in which a program in a host OS or a VMM detects malware by examining the data or behavior of the guest OS [2, 3, 4]. We expect that such a system would be combined with HyperSlow. For example, using a technique proposed in the research of Lycosid [4], the VMM administrator can find a process that consumes CPU cycles intensively in an almost OS-independent manner.

3.2 Timing Control

A process scheduler in an OS kernel usually performs process scheduling in response to timer interrupts. The scheduler checks whether the current process has consumed the entire time slice. The scheduler performs a context switch to another process if it has consumed. The process scheduler in Linux calculates the elapsed time using the number of timer interrupts and the values of hardware clocks that are acquired via the clocksource abstraction in recent kernels.

We next explain the behavior of a process scheduler with a sample case. Figure 2 (upper) portrays the CPU usage in this case. Processes A and B are running in the same domain and one VCPU is assigned to the domain. We assume that virtual timer interrupts are sent to the domain at every t ms and that the process scheduler in the guest kernel schedules the processes sequentially at every $4t$ ms (i.e., the time slice of processes is $4t$ ms). When a virtual timer interrupt is sent, the process scheduler checks whether the time slice has elapsed since the last scheduling. The scheduler obtains the value of system time at every virtual timer interrupt and calculates the (virtual) elapsed time from the last scheduling.

Presuming that a security system finds a malware process in a domain and notified our system about it, the hypervisor then sets the interval to $0.5t$ ms only while the malware process is running. In addition, the hypervisor modifies the value of (virtual) timer hardware

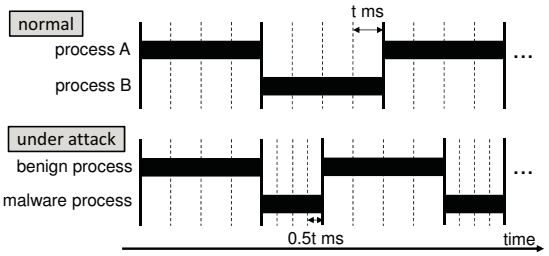

Figure 2: Changes in scheduling caused by the proposed method. Black areas represent the time during which the corresponding process is scheduled. Dashed lines represent the timing of timer interrupts.

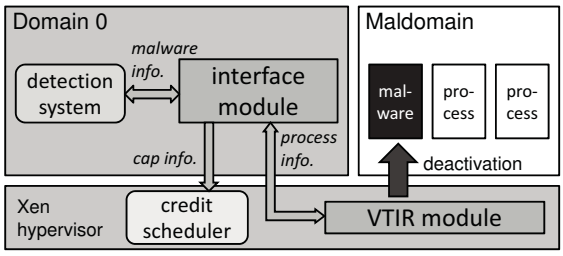

Figure 3: Structure of HyperSlow.

to create the illusion that t ms have elapsed since the last timer interrupt. The change is presented in Figure 2 (lower). Consequently, the virtual time in the domain passes at twice the original speed only while the malware process is running. The guest OS kernel miscounts the CPU time consumed by the malware process and therefore performs a context switch in $2t$ ms (half of the original time slice).

The administrator of the hypervisor might hope that the PCPU time deprived from the malware is restored not only to the maldomain but also in other domains. However, with a special mechanism, the deprived PCPU time is restored only to processes in the maldomain. Therefore, HyperSlow deprives the PCPU time of a maldomain by dynamically adjusting the maldomain cap.

The proposed method entails a side effect. It manipulates the virtual time. Therefore, the current time recognized by the guest OS kernel is later than the actual one. However, the side effect is normalized by Xen rapidly. The original Xen hypervisor periodically obtains a value from the actual hardware clock and writes the correct system time in the variable of system time. As a result, the current time of a guest OS gains temporarily and is then corrected. The user of the proposed method must accept that the current time returned by the guest kernel (e.g., result of `gettimeofday`) becomes incorrect when malware deactivation is working. Investigating the side effects of timekeeping subsystems such as ntpd and RADclock is an important topic for future work [1, 6].

3.3 Implementation Details

Figure 3 shows the system structure, which comprises the *interface module* and the *virtual timer interrupt raising (VTIR) module*.

The interface module receives alerts from a detection system. The alerts include the value of the page table register in the context of the malware process. The interface module transmits the value to the VTIR module via the shared memory between dom0 and the hypervisor. The interface module also determines a new cap of the maldomain and notifies the credit scheduler.

The VTIR module executes actual operations for slowing malware. The module shortens the interval between the virtual timer interrupts delivered to the VCPU associated with the maldomain. It further emulates the rate of time passage by modifying the variable of system time.

4 Experimental Results

4.1 Settings

We conducted experiments to confirm the practicality of the proposed method. We used an

Intel Core 2 Duo 2.53 GHz, 2 GB memory computer. 512 MB memory was assigned to each domain. We ran a paravirtualized version of Debian lenny (Linux 2.6.26) in domU's. The physical and virtual timer interrupt intervals were both 10 ms. The time slice of a domain was 30 ms and the default time slice of processes in a guest OS was 100 ms.

4.2 DoS Malware in an Ideal Setting

The first experiment was conducted in a setting that is regarded as ideal for HyperSlow. We executed one dom0 and one domU on the same hypervisor. We ran three processes in the domU: A, B, and M. The VCPUs of the domUs were pinned to one PCPU. All processes attempted to consume all CPU time. Processes A and B were regarded as benign programs such as scientific computations, although process M was regarded as malware performing DoS attacks against CPU resources.

All processes wrote a message to the standard output every time they completed a certain computation. We estimated the speed of each process from the output.

HyperSlow attempted to change the speed of process M to 1/1000 of the original speed. The quantities of completed computation by processes A, B, and M were, respectively, 301803, 305103, and 78. Process M became markedly slower than process A or B. The slowing of the malware was not 1/1000, but 1/3869. A likely reason is that the overhead of OS noise, including context switches, stood out because the actual time slice became extremely small.

4.3 DoS Malware in Virtual Hosting

We created an environment for hosting virtual servers. Then we executed one benign domain, one maldomain, and dom0 on the hypervisor. Each domain had one VCPU. The benign domain and maldomain shared one PCPU. We executed the Apache Web server in the benign domain. Then we executed, in the maldomain, malware that attempts to consume all CPU time.

Table 1: Benchmark result.

	average response time (ms)	worst response time (ms)
Original	8.52	3008
Attacked	76.63	46987
Deactivated	11.92	9004

We executed the ApacheBench benchmark and sent requests to the Web server from another physical machine connected with a gigabit ethernet switch. The benchmark requested a 3068-byte file repeatedly. The number of requests was 50,000 and the concurrency level was 25. We compared the following cases:

Original Malware was not running.

Attacked Malware was running and HyperSlow was not working.

Deactivated Malware was running and HyperSlow was working against the malware.

HyperSlow in this experiment varied the timer interrupt interval to 1/1000 and the speed of timer hardware to 1000 times.

Table 1 shows the response time of the Web server reported by the benchmark. When HyperSlow was not running, marked performance degradation was imposed on the response time. In contrast, when HyperSlow was working, the degradation was greatly reduced. Only 39.8% overhead was imposed on the average response time.

4.4 Spam-Mailing Malware

Finally, we created an environment in which malware in a maldomain sends spam e-mails. The malware accesses an external SMTP server on a different machine. It then requests the delivery of e-mails. The malware attempts to send an e-mail message every five seconds.

First, we measured the malware behavior when HyperSlow was not running. We confirmed that e-mails were sent every five seconds.

Then, we attempted to change the malware speed to 1/1000 of the original speed. Subsequently six e-mails were sent during 60 s. HyperSlow decreased the rate of mailing to half of the original. Usually, few CPU resources are consumed by spam-mailing malware, which is I/O-intensive and/or sleeping most of the time. Nevertheless, HyperSlow was able to slow such a program to some degree.

5 Related Work

FoxyTechnique [7] is a VMM-based technique that modifies the resource management policy of a guest OS by changing the behavior of virtual devices. Although their paper [7] describes the idea of changing the rate of timer interrupts, it shows no method of applying the technique to slow a process.

FoxyLargo [8] is a VMM-based mechanism that controls the speed of a virtual CPU with a fine granularity. FoxyLargo slows a whole virtual machine and invariably changes the speed of a particular process.

Many reports have been published about VMM-based security systems. Previous studies [2, 3, 4] specifically examined the development of malware detection, not malware prevention. Lycosid [4] is a VMM-based security system that is used to find hidden processes including stealth malware. It introduces a technique that enables a VMM to increase the execution time of specific processes by patching the code of the processes. They do not present an idea of applying timing control to the degradation of malware execution.

6 Summary and Future Work

We have proposed a method and system for extreme slowing of a malware process in a guest OS by varying the behavior of virtual hardware related to time management. Experimental results indicate that the method is useful for preventing damage from CPU-DoS attacks in VMM-based virtual hosting.

Future research might follow several paths. The first is investigation of the side effects on time management of a guest OS. The second is to combine HyperSlow with malware detectors proposed in other research. The third is evaluation of the method using other environments.

Acknowledgment This research was supported in part by KAKENHI 23700032.

References

[1] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch. Virtualize Everything but Time. In *Proc. of OSDI 2010*, pp. 451–464, 2010.

[2] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of NDSS 2003*, 2003.

[3] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection and Monitoring through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *ACM Transactions on Information and System Security*, 13(2):12:1–12:28, 2010.

[4] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *Proc. of VEE '08*, pp. 91–100, 2008.

[5] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proc. of the 2006 USENIX Annual Technical Conference*, pp. 1–14, 2006.

[6] VMware. Timekeeping in VMware Virtual Machines. `http://www.vmware.com/vmtn/resources/238`, 2011.

[7] H. Yamada and K. Kono. FoxyTechnique: Tricking Operating System Policies with a Virtual Machine Monitor. In *Proc. of VEE 2007*, pp. 55–64, 2007.

[8] T. Yoshida, H. Yamada, and K. Kono. FoxyLargo: Slowing Down CPU Speed with a Virtual Machine Monitor for Embedded Time-Sensitive Software Testing. In *Proc. of 2008 International Workshop on Virtualization Technology*, 2008.