A Self-protection Mechanism against Stepping-stone Attacks for IaaS Clouds

Kenichi Kourai
Kyushu Institute of Technology
kourai@ci.kyutech.ac.jp

Takeshi Azumi
Tokyo Institute of Technology
azumi@csg.is.titech.ac.jp

Shigeru Chiba
The University of Tokyo
chiba@acm.org

***Abstract*—For Infrastructure-as-a-Service (IaaS) clouds, stepping-stone attacks via hosted virtual machines (VMs) are critical. This type of attack uses compromised VMs as stepping stones for attacking the outside hosts. Not only compromised VMs but also IaaS providers are regarded as attackers. For self-protection, IaaS clouds should perform active response against stepping-stone attacks. However, it is difficult to stop only outgoing attacks at edge firewalls of clouds because edge firewalls can use only information in network packets. In this paper, we propose a new self-protection mechanism against stepping-stone attacks for IaaS clouds, which is called *xFilter*. xFilter is a packet filter running in the virtual machine monitor (VMM) underlying VMs and achieves pinpoint active response by using *VM introspection*. VM introspection enables xFilter in the VMM to obtain information on packet senders directly from the memory of VMs. When xFilter detects outgoing attacks, it automatically generates appropriate filtering rules with information on sender processes. Our experiments showed that xFilter could stop only outgoing attacks as much as possible. The performance degradation due to xFilter was less than 13 % in usual cases.**

***Keywords*-Virtual machines, operating systems, cloud computing, packet filtering, outgoing attacks**

I. Introduction

Infrastructure as a service (IaaS) such as Amazon EC2 [1] provides virtual machines (VMs) for the users. The users set up their own operating systems and applications in the VMs. Unfortunately, there is no guarantee that the systems inside VMs are well maintained. All security patches are not always applied to all software in all VMs. If the outside attackers compromise VMs in IaaS clouds, they can mount attacks to the outside hosts by using the VMs, which is known as *stepping-stone attacks* [2]. For example, the attackers may perform portscans to the outside hosts to find new victims. They may launch denial-of-service attacks.

Therefore, self-protection against such attacks is indispensable for IaaS clouds. If a VM in an IaaS cloud is used as a stepping stone for attacking the outside hosts, not only the user of the compromised VM but also the IaaS provider itself may be responsible for the attack. The IaaS provider also becomes an attacker as well as a victim. If an outgoing attack is detected, the IaaS cloud should perform *active response* against the attack. One of the methods for active response is updating firewall rules. Typically, a new rule for preventing the detected attack is added to the firewalls located at the edge of the IaaS cloud. The rule blocks the packets for the attack from the compromised VM and stops the attack against the outside hosts.

A self-protection mechanism for IaaS clouds should stop only attacks outgoing from compromised VMs. However, active response usually performed at edge firewalls is not *pinpoint* because edge firewalls can filter packets on the basis of only information contained in the packets. For example, edge firewalls would have to block all the packets from the compromised VM to stop portscans. Even when only several applications or users have been compromised, all the applications and users cannot send any packets to the outside. If IaaS clouds could use personal firewalls inside VMs, pinpoint active response could be achieved by using information inside the guest operating systems such as processes and users. This is not an option for IaaS providers because it is difficult to enforce such cooperation on the users.

To solve this dilemma, we propose a new self-protection mechanism, named *xFilter*, for IaaS clouds. xFilter is a packet filter that runs in the virtual machine monitor (VMM) underneath VMs and achieves pinpoint active response by using *VM introspection* [3]. VM introspection enables xFilter to inspect the memory of VMs and obtain information in guest operating systems without interacting with them. Using information on sender processes, for example, xFilter can deny only packets sent from particular processes or users. When xFilter detects an outgoing attack, it automatically identifies the attack source and generates a new filtering rule to stop the stepping-stone attack.

We have implemented xFilter in Xen [4]. xFilter is performance-critical because it performs VM introspection in the middle of packet transmission. To reduce the overheads of VM introspection, we introduced several optimizations. Although VM introspection is often implemented in the privileged VM called domain 0 [5], [6], we embedded the component for introspecting VMs into the VMM. The VMM can directly access the memory of VMs without any overheads. In addition, xFilter has the decision cache, which allows applying the same filtering decision for the packets flowed in the same TCP connection. Thanks to these optimizations, performance degradation due to xFilter was less than 13 % in usual cases.

The rest of this paper is organized as follows. Section II describes the issues of previous self-protection against

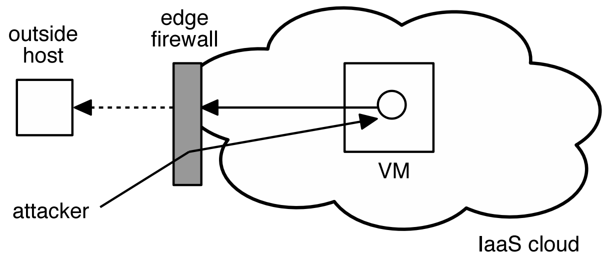

Figure 1. Self-protection using edge firewalls.

stepping-stone attacks. Section III presents xFilter, which is a new self-protection mechanism using VM introspection. Section IV explains its implementation based on Xen and Section V shows our experimental results for the effectiveness. Section VI examines related work and Section VII concludes the paper.

II. Self-protection against Stepping-stone Attacks

A self-protection mechanism for IaaS clouds should stop only stepping-stone attacks via VMs. In other words, VMs should provide services continuously as much as possible. Even if a part of the system inside a VM is compromised, the rest is usually not compromised and does not mount outgoing attacks. For example, when the Apache web server is compromised through its vulnerability, only the privileges of the user www-data are taken over at worst. The other applications such as the Postfix mail server are still running legitimately because the user www-data cannot interfere with the other users' processes. Therefore, such legitimate applications should be able to communicate with the outside hosts. Here we define applications that do not perform outgoing attacks as *legitimate*. In addition, if a self-protection mechanism often stops the communication of legitimate applications by false positives, the users would change their IaaS providers. At worst, they might sue the providers.

In IaaS clouds, packet filtering at edge firewalls is often performed for active response against stepping-stone attacks, as illustrated in Figure 1. However, it is difficult to deny only outgoing packets used for stepping-stone attacks at edge firewalls because edge firewalls can use only the information included in the network packets, such as IP addresses and port numbers. Packet filtering based on source IP addresses is the simplest active response. If an IaaS cloud adds only one rule for denying all the packets from a compromised VM, it can completely prohibit outgoing attacks from the VM. This active response is reasonable when the whole system is compromised. When the system is partially compromised, legitimate applications in the VM cannot also send any packets to the outside.

Using more information in packet headers enables more pinpoint active response, but it is still difficult to stop only stepping-stone attacks. Packet filtering based on destination IP addresses can deny packets sent to a specified host while it can allow packets to be sent to the others. Adding such filtering rules to edge firewalls is reasonable if stepping-stone attacks are mounted only to a small number of target hosts. It is not realistic to add a large number of rules. For example, the intruders may perform SMTP scans to many hosts to find vulnerable mail servers. In such a case, packet filtering based on destination port numbers can prohibit sending packets to only specific services at all hosts. By port 25 blocking, the intruders cannot perform SMTP scans to any hosts. However, legitimate applications cannot send e-mail as well. Even if intrusion detection systems in the VMs detect intrusion and send alert mails to the administrators of the VMs, those e-mails would be blocked at edge firewalls.

Using information on source port numbers is promising for pinpoint active response. It enables edge firewalls to prohibit only particular network connections. For example, if edge firewalls detect illegal TCP connections, they can add filtering rules and block only those connections. Unfortunately, specifying source port numbers is too fine-grained. Although such rules are effective for long-lived network connections such as SSH, they are useless for short-lived connections such as portscans. When filtering rules are added to edge firewalls, those short-lived connections would have been already closed.

On the other hand, using personal firewalls inside VMs can achieve appropriately pinpoint active response. Personal firewalls can use information on sender processes for packet filtering because they reside in the operating system kernels. For example, iptables [7] in Linux and ipfw in FreeBSD allow process IDs and user IDs of packet senders as a part of filtering rules. They can block outgoing packets only when attacks are mounted by processes or users that are taken over by the intruders. Therefore, legitimate applications and users can communicate with the outside hosts. However, IaaS providers usually have no privileges for adding rules to the personal firewalls in VMs. Although they have to cooperate with the administrators of VMs, all of them are not always cooperative.

III. xFilter

For appropriately pinpoint active response against stepping-stone attacks, we propose a new self-protection mechanism for IaaS clouds, named *xFilter*. xFilter automatically detects outgoing attacks and stops only them on the basis of information on packet senders. In this paper, we do not assume that the attackers alter the operating system kernels in VMs. This type of attack can be detected by the VMM [3], [8].

A. VMM-level Packet Filtering

xFilter is a packet filter running in the VMM as in Figure 2. The VMM is underlying software that runs VMs

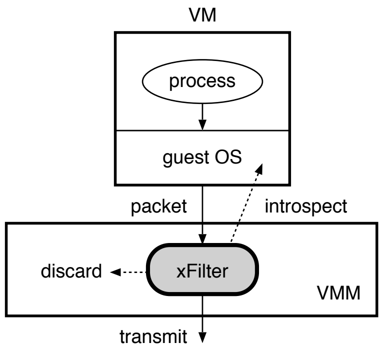

Figure 2. xFilter running in the VMM.

on top of it. xFilter can intercept all network packets from VMs because all packets are transmitted to the outside via the VMM. Unlike edge firewalls, xFilter can also intercept packets between VMs even in the same host. This prevents stepping-stone attacks via one VM to another in a host. Moreover, xFilter is isolated and protected from all the VMs. It is difficult for the intruders in VMs to compromise xFilter in the VMM.

To stop only outgoing attacks from VMs, xFilter uses information inside guest operating systems by using *VM introspection* [3]. In principle, the VMM cannot know such information because it is unaware of the internals of VMs. In this sense, the VMM is similar to edge firewalls. One of the differences is that the VMM can directly access the memory of VMs. VM introspection is a technique for enabling the VMM to inspect data used by guest operating systems. It analyzes the memory of VMs on the basis of the information on the internal structures of guest operating systems.

Using VM introspection, xFilter obtains information on packet senders, such as process IDs and user IDs, from guest operating systems. For each packet, it searches a network socket used for sending the packet on the basis of IP addresses and port numbers. A process that opens the found socket is the sender process of the packet. The owner of the process is the user sent the packet. As such, xFilter in the VMM can achieve appropriately pinpoint active response as personal firewalls in VMs can. For example, xFilter can block only the packets from processes used for stepping-stone attacks by specifying their process IDs or user IDs.

The packet filtering in xFilter is performed as follows. When a process sends a packet, the guest operating system writes it to a virtual network interface card (NIC). The VMM traps that event and passes the packet to xFilter. xFilter obtains the process ID and user ID of the packet sender through VM introspection. If the destination and the sender of the packet match one of the filtering rules, xFilter discards the packet. Otherwise, it transmits the packet to the network.

```
deny ip * port * vm 1 pid 1234 uid 501
deny ip * port * vm 1 pid *    uid 501
```

Figure 3. The rules added by xFilter for preventing portscans.

B. Automatic Rule Generation

To achieve self-protection of IaaS clouds, xFilter automatically generates filtering rules when it detects outgoing attacks from VMs. The detector of xFilter has two phases: detection and inspection. In the detection phase, xFilter examines outgoing packets with only information included in packet headers. Since the detector of xFilter in this phase is the same as the ones for edge firewalls, the overhead of the attack detection is minimum.

Once xFilter detects an attack, it changes into the inspection phase. In this phase, whenever xFilter receives a packet, it identifies the sender process from the packet information by using VM introspection. When xFilter detects an attack again, it generates a *deny* rule that consists of the IP address and port number of the destination host, the ID of the source VM, and the process ID and user ID of the attack source. For example, when a process whose ID is 1234 and owner's user ID is 501 performs portscans against various hosts, xFilter generates the first rule in Figure 3. Since this rule is process-level, the other processes can send packets. The rule is effective as long as the specified process continues portscans.

To increase the effectiveness of the generated rules, xFilter merges a generated rule with the existing ones as necessary. If the attack source changes frequently, process-level rules become ineffective soon. When there are many rules in which only process IDs are different, xFilter merges them into one new rule in which the process ID is a wildcard, as shown in the second rule in Figure 3. This rule is user-level and effective as long as the specified user continues portscans. However, this is coarser-grained than process-level rules, so that the specified user cannot send any packets. If the root privileges are taken, the attackers can change even user IDs frequently as well. In this worst case, xFilter generates one new rule in which the user ID is also a wildcard. Since this rule specifies neither process ID nor user ID, it becomes the same rule as one used at edge firewalls.

C. Limitation

xFilter can prevent only outgoing attacks such as portscans, SMTP scans, brute force attacks against SSH, DoS attacks, worms, and so on. Filtering rules are different only in how to specify the destination IP address and port number. However, xFilter cannot perform pinpoint active response when the attackers and legitimate applications use server processes shared in a VM to send packets. For example, the attackers can use a local SMTP server to mount SPAM attacks whereas the other applications use the same

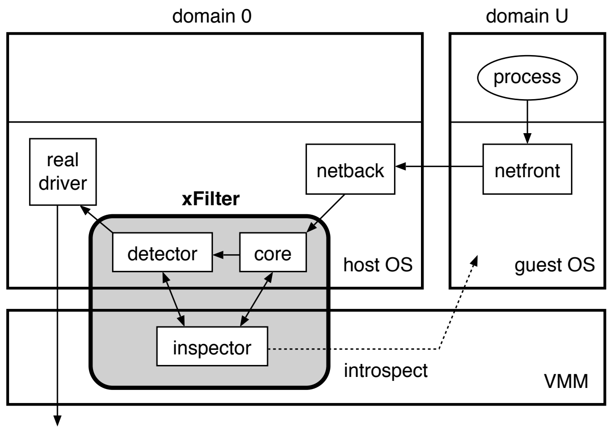

Figure 4. The architecture of xFilter in Xen.

server. When xFilter detects the SPAM attacks, it adds a rule for denying all the packets from the SMTP server, which is regarded as the process of an attack source. This rule blocks e-mails from not only the attackers but also legitimate applications. To achieve pinpoint active response, applications should use external SMTP servers to send e-mails. Specifically, Perl and PHP scripts in web servers should use Net::SMTP and PEAR::Mail, respectively. If applications do not use a local SMTP server, xFilter could block e-mails on the basis of the sender processes.

IV. Implementation

We have implemented xFilter in Xen 3.4.2 [4]. Xen provides a privileged VM called *domain 0* and regular VMs called *domain Us*. Domain 0 is often regarded as a part of the VMM because it handles I/O for domain Us. We targeted para-virtualized Linux 2.6.18 for the x86-64 architecture as guest operating systems running in domain Us.

A. System Architecture

As illustrated in Figure 4, xFilter consists of three components: the core, the detector, and the inspector. When a process issues system calls such as `send` in domain U, the operating system kernel in domain U transmits a packet with the front-end network driver called *netfront*. The netfront driver passes the packet to the back-end driver called *netback* in the kernel of domain 0. Then the *netback* driver invokes the *xFilter core*, instead of invoking a real network driver. If the core decides to deny sending that packet, it discards the packet. Otherwise, it passes the packet to the *xFilter detector*. If the detector judges that the packet is used for attacks, it generates a new filtering rule and discards the packet. If the packet is not for attacks, the detector passes the packet to the real driver and the driver transmits it to the network.

When the xFilter core needs information on packet senders to decide whether packets are permitted for transmission, it invokes the *xFilter inspector* in the VMM. It issues a hypervisor call to the VMM and the hypervisor call takes the ID of the source domain U and the packet header as its parameters. The inspector introspects domain U to identify the packet sender and makes a decision on packet filtering with sender information. Then the hypervisor call returns the decision to the core. Also, the xFilter detector invokes the inspector to identify the attack source in the inspection phase.

The reason for running only the xFilter inspector in the VMM is efficiency. Although the xFilter core and detector in domain 0 could introspect domain U by mapping its memory pages, the overhead is larger. The VMM can directly access the memory of domain U because it manages the whole memory in the system. In addition, xFilter can handle all packets only in domain 0 until stepping-stone attacks are detected. While it has no filtering rules, the core can immediately pass packets to the detector without invoking the inspector in the VMM. If the all components of xFilter ran in the VMM, the netback driver would have to always issue the hypervisor call to the VMM.

B. VM Introspection

To access an object in the guest operating system from the outside of domain U, the xFilter inspector has to translate virtual addresses in domain U to *machine addresses*. In Xen, the VMM uses the machine address to access the whole memory. Domain U is given *pseudo-physical memory* for the illusion of its own physical memory. First, the inspector looks up the page tables in domain U and translates a virtual address to a pseudo-physical address in the domain U. Next, it looks up the *P2M table* in the VMM and translates the pseudo-physical address to a machine address. The P2M table maintains the mapping from pseudo-physical frame numbers to machine frame numbers, which are consecutively allocated for memory pages.

The xFilter inspector uses the debug information of the operating system kernels to obtain information on the data structures and the addresses allocated to global symbols in guest operating systems. For example, in Linux, the task_struct structure contains process information such as its ID and owner's user ID. The init_task symbol points to the task_struct object for the init process. xFilter can obtain such information from its kernel image compiled with the debug option. Such a kernel image contains debug information in the DWARF [9] format.

Figure 5 illustrates how the xFilter inspector traverses kernel data structures to find a process sending a particular packet. The inspector first traverses the process list in domain U, which consists of all task_struct objects. Since the process list is circular, the inspector can examine all the processes. While traversing the process list, the inspector deeply inspects the socket list that each process owns. If the inspector finds the inet_sock object whose source and destination IP addresses and port numbers match the

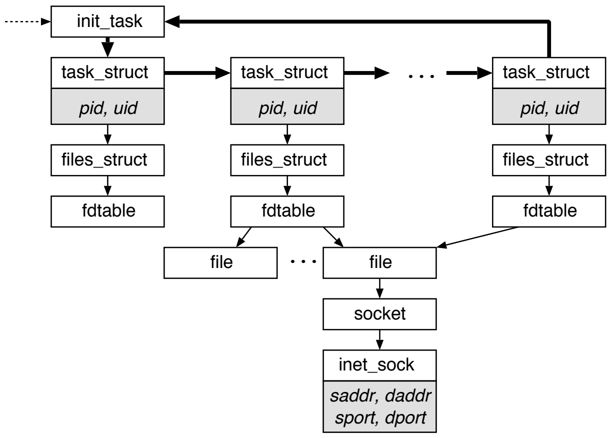

Figure 5. The traversal of kernel data structures.

target packet, it regards the process owning that object as the sender. When one socket is shared between multiple processes by spawning the process that created the socket, the inspector regards the original process as the sender. This approach is the same as that in iptables [7]. Since spawned processes are appended to the tail of the process list, the inspector can easily find such a process by traversing the list from the head.

If an appropriate inet_sock object is not found, a *raw socket* is probably used for sending the packet. Raw sockets enable the user to assemble packets with protocol headers. Therefore, the inet_sock objects for raw sockets do not have correct information on IP addresses and port numbers. In this case, the xFilter inspector regards all the processes that open any raw sockets as the senders.

The xFilter inspector can optimize this traversal when it decides whether a packet matches one of the filtering rules or not. While the inspector traverses the process list, it checks whether the ID or owner of each process matches one of the filtering rules. If both do not match any rules, the inspector can skip the deep traversal of the socket list opened by the process. Only for a process whose ID or owner matches at least one of the rules, the inspector inspects sockets that the process opens.

To introspect the guest operating system consistently, the xFilter inspector checks whether the guest operating system acquires locks for manipulating data structures. For the process list, the inspector examines the spin lock used for atomically adding and removing a process. If the spin lock is not acquired by the guest operating system, the inspector can traverse the process list safely. It does not need to acquire that spin lock for exclusive access because domain U is paused during VM introspection and it does not compete for the lock. If the guest operating system acquires that lock, the inspector aborts VM introspection and attempts to handle that packet after a while.

C. Decision Cache

The xFilter core has *decision cache* for reducing the overhead of VM introspection. The decision cache stores the decisions made by the xFilter inspector. Packets flowed in the same connection hit on the decision cache. In this case, the core reuses the decision obtained from the decision cache instead of invoking the inspector. Even if the core invokes the inspector, it would obtain the same decision as the cached one in most cases. When a sender process changes its owner, the latest decision by the inspector may be different from the cached one. However, xFilter applies the cached decision because packets in the same connection should be related to the original process and owner.

For TCP connections, the xFilter core manages the decision cache based on the TCP control bits in packet headers. When the core receives a packet with the SYN flag set, it invokes the xFilter inspector and then adds a new entry to the decision cache. The SYN flag is set when a new connection is being established. The entry includes source and destination IP addresses and port numbers as a packet information and *allow* or *deny* as a decision. When the core receives a packet with the FIN or RST flag set, it removes the corresponding entry. The FIN flag is set when an existing connection is terminated while the RST flag is set when a connection is reset. For the other packets in which the above flags are not set, the core looks up the decision cache. The decision cache manages its cache entries in a least-recently-used (LRU) manner. If there is no matching entry, the core simply invokes the inspector.

V. Experiments

We performed experiments for demonstrating the effectiveness of xFilter and examining its overheads. Since VM introspection used by xFilter is time-consuming, it can affect the network performance largely. For a server machine, we used a PC with one Intel Core i7 processor 860, 8 GB of memory, and a Gigabit Ethernet NIC. The VMM was Xen 3.4.2 and the guest operating systems in domain 0 and domain U were Linux 2.6.18. We allocated 7 GB of memory to domain 0 and 1 GB to domain U. For a client machine, we used a PC with one Athlon 64 processor 3500+, 2 GB of memory, and a Gigabit Ethernet NIC. These two machines were connected with a Gigabit Ethernet switch.

A. Self-protection against Portscans

To demonstrate that xFilter enables self-protection against portscans, we have implemented a portscan detector as the xFilter detector. The detector records the packet headers and timestamps and detects portscans if packets were sent to many ports at an excessive rate. Then we performed portscans from a victim VM to the outside hosts using nmap. We attempted both normal TCP scans using regular sockets and TCP SYN scans using raw sockets. First, we ran one nmap process in the VM. As a result, xFilter could detect

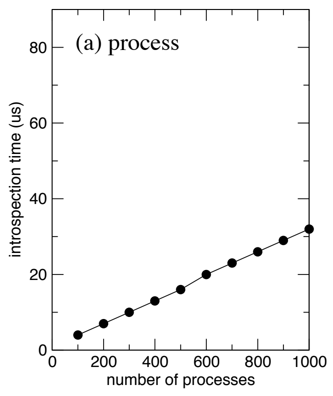

Figure 6. The introspection time for the various numbers of processes and sockets.

the both types of portscans and stop the successive attacks by automatically generating a filtering rule like the first rule in Figure 3. We confirmed that the other processes such as SSH could communicate with the outside hosts under this rule.

Next, we ran many nmap processes in the VM by starting another nmap process after one nmap process finished a sequence of portscans. In this case, xFilter also detected the portscans and generated filtering rules like the first rule in Figure 3 for each process. After we continued the portscans, xFilter automatically merged these rules into one rule like the second rule in Figure 3 to stop any portscans from the same user. The user could not communicate with any outside hosts due to this rule, but the other users could still use the network.

B. Overheads of VM Introspection

To examine the overhead of VM introspection, we measured the time needed for executing the xFilter inspector. First, we changed the number of processes that xFilter inspected and measured the execution time. We specified a non-existent process ID in a filtering rule so that the xFilter inspector traversed the entire process list and checked the process IDs of all the processes. In this experiment, the inspector did not deeply inspect kernel data structures for sockets. Figure 6 (a) shows the execution time, which is proportional to the number of processes and takes 31 ns per process.

Second, we changed the number of sockets that xFilter inspected and measured the execution time of the xFilter inspector. We specified an existent user ID in a filtering rule so that the inspector deeply inspected sockets. Figure 6 (b) shows the result. The time is approximately proportional to the number of sockets and takes 83 ns per socket. This means that the overhead of inspecting sockets is larger than that of inspecting processes.

Third, we change the number of filtering rules for xFilter and measured the execution time of the xFilter inspector. We

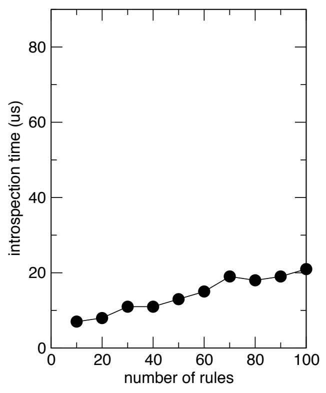

Figure 7. The introspection time for the various numbers of rules.

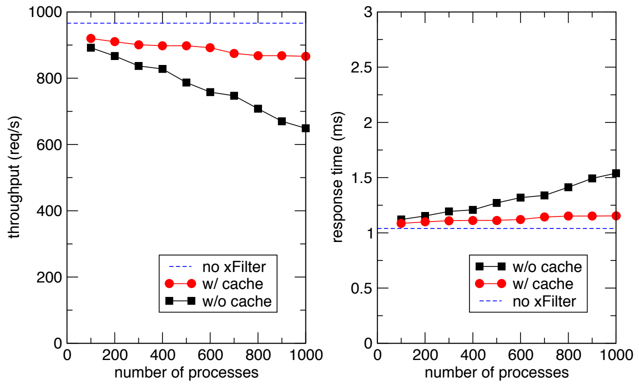

Figure 8. The web performance for the various numbers of processes.

specified a non-existent process ID in all the rules. Figure 7 shows the result. The time is proportional to the number of rules and takes 160 ns per rule. The number of rules is usually not so large because xFilter merges rules.

C. Filtering Performance

To examine performance degradation in a real application, we measured the throughput and the response time of the Apache web server [10]. We used the ApacheBench benchmarking tool, which ran in the client machine. ApacheBench sent HTTP requests to the web server running in the VM of the server machine. The size of the requested HTML file was 50 KB. We conducted three experiments for various numbers of processes, sockets, and rules. For each, we used the same filtering rules as in the above experiments. We measured the performance when the decision cache was disabled and enabled to show its effectiveness. When we did not use xFilter, the throughput was 966 requests/s and the response time was 1.04 ms.

First, we measured the web performance when we changed the number of processes. Figure 8 shows the throughput and response time, which degrade in proportion to the number of processes. When the decision cache was enabled, the throughput and response time degraded by 7 %

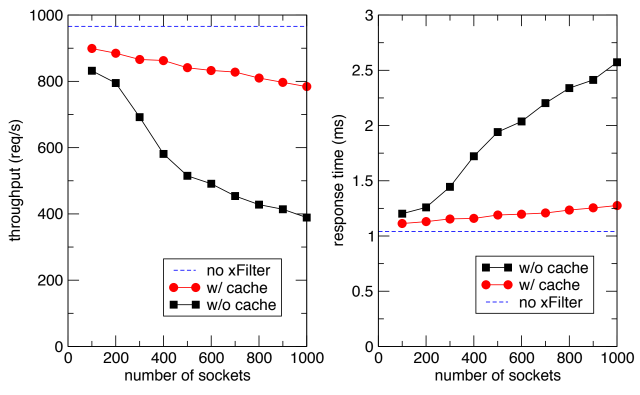

Figure 9. The web performance for the various numbers of sockets.

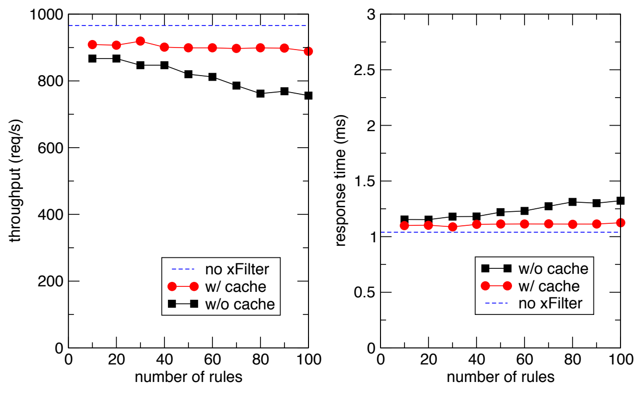

Figure 10. The web performance for the various numbers of rules.

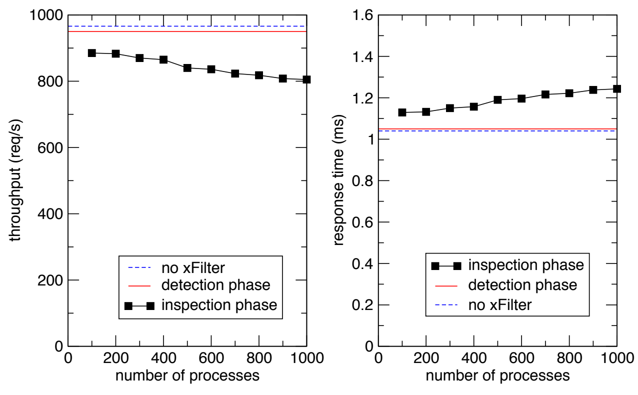

Figure 11. The performance for the various numbers of target processes.

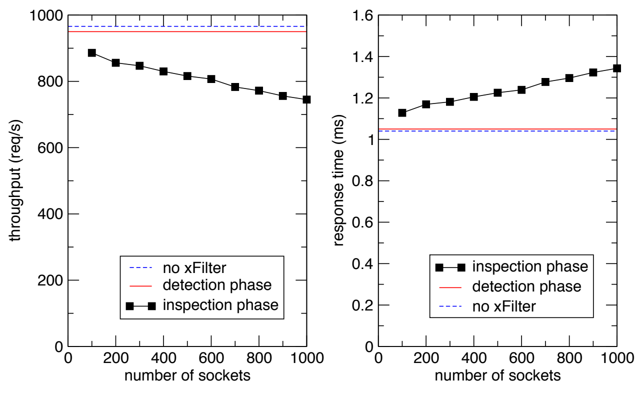

Figure 12. The performance for the various numbers of target sockets.

for realistic 300 processes. On the other hand, with the decision cache disabled, the throughput was degraded by 13 % and the response time increased by 15 %. From these results, the decision cache is effective for reducing the overheads.

Next, we measured the web performance for various numbers of sockets. The throughput and response time are shown in Figure 9. The performance degradation is proportional to the number of sockets. With the decision cache enabled, the throughput and response time degraded by 11 % for realistic 300 sockets. When the decision cache was disabled, however, the throughput degradation was 28 % and the response time was 39 % longer. This shows that the number of sockets affects the web performance more largely.

Third, we changed the number of filtering rules and measured the web performance. As shown in Figure 10, with the decision cache enabled, the throughput degradation was 11 % and the response time increased by 9 % even for 100 rules. In reality, if 100 rules are needed, the system would be completely compromised.

D. *Overheads of Attack Detection*

To examine the overheads of only attack detection by the xFilter detector, we measured the performance of the web server when xFilter had no filtering rules yet. Figure 11 and Figure 12 show the web performance when we changed the number of processes and sockets, respectively. In the detection phase, the performance degradation was only 1 % because xFilter did not need VM introspection. This means that the overhead of the xFilter detector is small enough until outgoing attacks are detected. In the inspection phase, on the other hand, the performance degraded as the numbers of processes and sockets increased. The performance degraded by about 11 % for 300 processes while it degraded by about 13 % for 300 sockets.

VI. Related Work

The most similar system to xFilter is VMwall [11], which is an application-level firewall using VM introspection. It uses information on processes sending or receiving packets for fine-grained packet filtering. One of the important differences is that VMwall performs VM introspection using a process in domain 0. According to our experience, this degrades the network performance severely, particularly, when large numbers of processes and sockets are to be inspected in domain U. Since xFilter performs VM introspection in the VMM, the performance degradation is minimized even in such a situation.

Intrusion detection systems using VM introspection have been proposed [3], [12]. They examine the internal state of the operating system kernel in a VM from the outside and detect attacks. One of the primary differences from xFilter is that they are not performance-critical so much. Livewire [3] is an offline detection tool while IntroVirt [12] is applied to intrusion detection in infrequent execution paths. Since xFilter is invoked during network packet transmission, its performance directly affects the network performance.

XenAccess [5] and LibVMI [13] are libraries for introspecting guest operating systems. They support translating virtual addresses used inside VMs and mapping the memory pages of VMs. xFilter did not use them because they intend to be used by user-level applications on domain 0 or the host operating system. The xFilter inspector that performs VM introspection runs in the VMM. Therefore, we have developed the mechanism for inspecting guest operating systems from the VMM.

Amazon EC2 provides firewalls called security groups [14] for VMs. Security groups are probably implemented in domain 0 of Xen. They are located in the outside of VMs, but they are provided to the IaaS users. It is uncertain whether the cloud provider adds filtering rules to security groups. In addition, security groups cannot prevent stepping-stone attacks because they are inbound-only firewalls against attacks from the outside. They cannot filter any packets from the inside. Also, they cannot use information on guest operating systems inside VMs.

The ident protocol [15] is defined for querying the user who sends a packet. When one host sends a pair of source and destination port numbers to the ident server running in the other host, the server returns the owner of a process that uses the specified network connection. However, edge firewalls in clouds cannot use this protocol to obtain information on packet senders because this protocol is designed to be used by end hosts. Moreover, the ident server may not be trustworthy when the host is compromised by stepping-stone attacks.

VII. Conclusion

In this paper, we proposed a new self-protection mechanism against stepping-stone attacks for IaaS clouds, which is called xFilter. For pinpoint active response, xFilter runs in the VMM and uses information on sender processes in compromised VMs for packet filtering. Using VM introspection, it directly obtains the process IDs and user IDs of sender processes in the VMs. Our experiments showed that the overhead of xFilter were only 1 % until stepping-stone attacks were detected. Even after the attack detection, the performance degradation of the web server was less than 13 % for usual numbers of processes and sockets.

One of our future work is supporting UDP in the decision cache. Like stateful packet inspection, the decision cache can manage UDP sessions as virtual connections. Another direction is using other information on sender processes for packet filtering. For example, grouping processes with information on their ancestors may be helpful for finer-grained filtering.

Acknowledgment

This research was supported in part by JST, CREST.

References

[1] Amazon, Inc., “Amazon Elastic Compute Cloud,” http://aws.amazon.com/ec2/.

[2] S. Staniford-Chen and L. Heberlein, “Holding Intruders Accountable on the Internet,” in *Proc. Symp. Security and Privacy*, 1995, pp. 39–49.

[3] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization,” in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[5] B. Payne, M. Carbone, and W. Lee, “Secure and Flexible Monitoring of Virtual Machines,” in *Proc. Annual Conf. Computer Security Applications*, 2007, pp. 385–397.

[6] B. Payne, M. Carbone, M. Sharif, and W. Lee, “Lares: An Architecture for Secure Active Monitoring Using Virtualization,” in *Proc. Symp. Security and Privacy*, 2008, pp. 233–247.

[7] Netfilter Core Team, “The netfilter.org Project,” http://www.netfilter.org/.

[8] J. N. Petroni and M. Hicks, “Automated Detection of Persistent Kernel Control-flow Attacks,” in *Proc. Conf. Computer and Communications Security*, 2007.

[9] DWARF Standards Committee, “The DWARF Debugging Standard,” http://dwarfstd.org/.

[10] Apache Software Foundation, “Apache HTTP Server Project,” http://httpd.apache.org/.

[11] A. Srivastava, , and J. Giffin, “Tamper-resistant, Application-aware Blocking of Malicious Network Connections,” in *Proc. Int. Symp. Recent Advances in Intrusion Detection*, 2008, pp. 39–58.

[12] A. Joshi, S. King, G. Dunlap, and P. Chen, “Detecting Past and Present Intrusions through Vulnerability-specific Predicates,” in *Proc. Symp. Operating Systems Principles*, 2005, pp. 91–104.

[13] B. Payne, “LibVMI,” http://code.google.com/p/vmitools/.

[14] Amazon, Inc., “Amazon Web Services: Overview of Security Processes,” http://aws.amazon.com/security/, 2009.

[15] M. Johns, “Identification Protocol,” RFC 1413, 1993.