Efficient and Fine-grained VMM-level Packet Filtering for Self-protection

Kenichi Kourai
Kyushu Institute of Technology

Takeshi Azumi
Tokyo Institute of Technology

Shigeru Chiba
The University of Tokyo

Abstract

In Infrastructure-as-a-Service (IaaS) clouds, stepping-stone attacks via hosted virtual machines (VMs) are critical for the credibility. This type of attack uses compromised VMs as stepping stones for attacking the outside hosts. For self-protection, IaaS clouds should perform active responses against stepping-stone attacks. However, it is difficult to stop only outgoing attacks at edge firewalls, which can only use packet headers. In this paper, we propose a new self-protection mechanism against stepping-stone attacks, which is called *xFilter*. xFilter is a packet filter running in the virtual machine monitor (VMM) underlying VMs and achieves pinpoint active responses by using *VM introspection*. VM introspection enables xFilter to directly obtain information on packet senders inside VMs. On attack detection, xFilter automatically generates filtering rules based on packet senders. To make packet filtering with VM introspection efficient, we introduced several optimization techniques. Our experiments showed that the performance degradation due to xFilter was usually less than 16 %.

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

INTRODUCTION

Cloud computing is rapidly emerging in recent years. Among various types of clouds, infrastructure as a service (IaaS) such as Amazon EC2 (Amazon, Inc., 2006) provides virtual machines (VMs) for the users. The users can use their VMs on demand. Unfortunately, it is not guaranteed that the systems inside VMs are always well-maintained. If the outside attackers compromise such VMs, they can mount attacks to the outside hosts via the VMs, which is known as *stepping-stone attacks* (Staniford-Chen & Heberlein, 1995). For example, the attackers may perform portscans and denial-of-service (DoS) attacks to the outside hosts.

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 VM's user but also the IaaS provider may have a responsibility for the attack. The IaaS provider also becomes an attacker as well as a victim. If an IaaS cloud detects outgoing attacks, it should perform *active responses* against the attacks. One of the methods for active responses is updating firewall rules.

Typically, new rules are added to edge firewalls in the IaaS cloud. The rules block the packets for the attacks from the compromised VM.

Such a self-protection mechanism should stop only outgoing attacks. However, active responses performed at edge firewalls are 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 the system is partially compromised, all the applications and users cannot send any packets to the outside. Pinpoint active responses are beneficial to not only IaaS users but also providers because IaaS providers can easily stop suspicious communication without excessive fears of false positives.

In this paper, we propose a new self-protection mechanism, named *xFilter*, for IaaS clouds. xFilter is a packet filter running in the virtual machine monitor (VMM), which is underneath VMs. To achieve pinpoint active responses, xFilter obtains information on packet senders by using a technique called *VM introspection* (Garfinkel & Rosenblum, 2003). VM introspection enables xFilter to inspect the memory of VMs and access data in guest operating systems without interacting with them. Using information on sender processes, 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. In addition, xFilter provides development support of its modules because it is difficult to develop software performing VM introspection in the VMM.

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 optimization techniques. First, we embedded the component for introspecting VMs into the VMM of Xen (Barham et al., 2003) although VM introspection is usually performed in the privileged VM (Jiang, Wang, & Xu, 2007; Payne, Carbone, & Lee, 2007; Payne, Carbone, Sharif, & Lee, 2008). Second, xFilter performs *optimized sender traversal* to find sender processes so that the number of kernel objects to be introspected is minimized. Third, xFilter provides the *decision cache* to reuse filtering decisions without VM introspection. Fourth, *two-phase attack detection* minimizes the overheads under no attack symptoms. Thanks to these techniques, performance degradation due to xFilter was less than 16 % in usual cases.

This paper is an extended version of our previous paper (Kourai, Azumi, & Chiba, 2012). In this paper, we describe four optimization techniques for xFilter explicitly and in further detail, particularly for optimized sender traversal and two-phase attack detection. In addition, we introduce development support for xFilter and its optimization, and we report the performance in the development phase. Moreover, we add new experiments for optimized sender traversal and raw sockets. We also explain the implementation details of our portscan detector.

BACKGROUND

For IaaS clouds, a fine-grained self-protection mechanism is needed to stop only stepping-stone attacks via VMs. In other words, VMs should be able to provide services continuously as much as possible. Even if the system inside a VM is partially compromised, the rest is usually not compromised. For example, when the Apache web server is compromised, only the privileges of the user www-data are taken over at worst. The other applications 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.

Such fine-grained self-protection is beneficial to not only IaaS users but also IaaS providers. For IaaS providers, it is safer to stop all communication from partially-compromised VMs when any attack is detected. However, there is a certain rate of false positives for attack detection because accurate attack detection is difficult, particularly for portscans and DoS attacks. If a self-protection mechanism often stops the communication of legitimate applications, the users would change their IaaS providers. At worst, they might sue the providers. Using fine-grained self-protection, IaaS providers can stop a part of communication without excessive fears of false positives. Then they can give a time for solving problems to the users.

Active Responses at Existing Firewalls

Packet filtering at edge firewalls is often performed for active responses against stepping-stone attacks. However, it is difficult to deny only outgoing packets used for stepping-stone attacks at edge firewalls. 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 is not reasonable for partially-compromised systems because even legitimate applications in the VM cannot send any packets to the outside.

Using more information in packet headers enables relatively pinpoint active responses. 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. Although this is reasonable only for a small number of 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 blocking port 25, the intruders cannot perform SMTP scans to any hosts. However, legitimate applications cannot send e-mail as well, e.g., alert mails for detected intrusions.

Using information on source port numbers is promising for pinpoint active responses. It enables edge firewalls to prohibit only particular network connections. 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 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 responses. Personal firewalls can use information on sender processes for packet filtering. For example, iptables in Linux allows 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. However, IaaS providers usually have no privileges for adding rules to the personal firewalls in VMs.

Related Work

Amazon EC2 provides firewalls called security groups (Amazon, Inc., 2009) for VMs. Security groups 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 (Johns, 1993) is defined for querying the user who sent a packet. When one host sends a pair of source and destination port numbers to the ident server, the server returns the owner of a process that uses the specified network connection. However, edge firewalls cannot use this protocol because this protocol is designed to be used between end hosts. Moreover, the ident server may not be trustworthy when the host is compromised by stepping-stone attacks.

The most similar system to xFilter is VMwall (Srivastava & Giffin, 2008), 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 of Xen. 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 and optimizes it with several techniques, the performance degradation is minimized even in such a situation. Another difference is that VMwall performs whitelist-based packet filtering and does not generate filtering rules dynamically. Therefore it does not provide autonomic self-protection.

ADVOS (Garg & Saran, 2008) prevents outgoing DoS attacks mounted by compromised VMs. It detects DoS attacks only using packet information and limits the rate of packet transmission in domain 0 of Xen. Since ADVOS does not use information on sender processes, it affects all the packets sent from compromised VMs. It cannot limit the transmission rate of only the packets used for DoS attacks. ADVOS proposes VM introspection to identify malicious processes mounting DoS attacks.

VM introspection has been also applied to other security systems such as intrusion detection (Garfinkel & Rosenblum, 2003; Joshi, King, Dunlap, & Chen, 2005; Payne et al., 2008) and malware analysis (Jiang et al., 2007; Dinaburg, Royal, Sharif, & Lee, 2008). These systems examine the internal state of the operating system kernel from the outside of the VM and detect attacks. One of the primary differences between them and xFilter is that they are less performance-critical than xFilter. Since xFilter is invoked during network packet transmission, its performance directly affects the network performance.

Like the development of xFilter running in the VMM, it is difficult to develop kernel modules of operating systems. Therefore, several operating systems such as Chorus (Rozier et al., 1992) and CAPERA (Kourai, Chiba, & Masuda, 1998) support the development of kernel modules. They allows developers to first implement kernel modules as user processes and then embed them into the kernel without any modifications. To the best of our knowledge, xFilter is the first work providing such development support for components of the VMM.

XFILTER

For appropriately pinpoint active responses 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 assume that the attackers compromise servers running inside VMs and take their privileges. Then they attack the outside hosts by sending packets from victim VMs. We do not assume that the attackers modify kernel data structures in VMs or replace the operating systems themselves with malicious ones. This type of attack can be detected by the VMMs (Garfinkel & Rosenblum, 2003; Petroni, Jr. & Hicks, 2007).

VMM-level Packet Filtering

xFilter is a packet filter running in the VMM as in Figure 1. The VMM is underlying software that runs VMs 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* (Garfinkel & Rosenblum, 2003). 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.

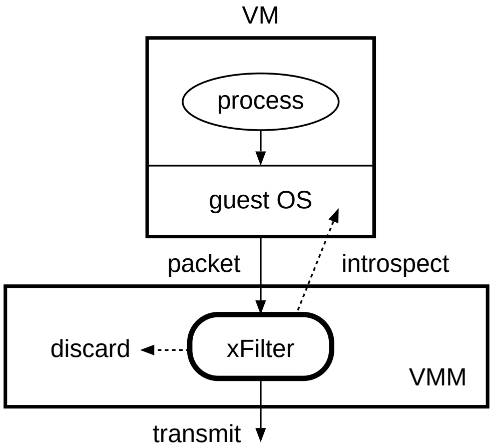


Figure 1. xFilter running in the VMM.

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

Figure 2. The rules added by xFilter for preventing portscans.

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, and the owner of the process is the user sent the packet. Using such information, xFilter can block only the packets sent from processes used for stepping-stone attacks. As such, xFilter in the VMM can achieve appropriately pinpoint active responses as personal firewalls inside VMs can.

Automatic Rule Generation

To achieve self-protection of IaaS clouds, xFilter automatically generates filtering rules. When it detects outgoing attacks from VMs, it identifies the sender process from the packet information by using VM introspection. Then 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 2. If attacks are mounted against a specific IP address or port, xFilter generates a rule that specifies it. Such a rule is process-level and effective as long as the specified process continues portscans. The generated rule is removed when any packets are not filtered out by the rule for a while, e.g., 30 minutes.

To increase the effectiveness of the generated rules, xFilter merges a generated rule with the existing ones as necessary. If an attacking process changes frequently, process-level rules become ineffective soon. When there are many generated rules, e.g., five 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 2. Such a rule is user-level and effective as long as the specified user continues portscans. Instead, this is coarser-grained than process-level rules, so that the specified user cannot send any packets. If the attackers take root privileges and change user IDs frequently, xFilter generates one new rule in which the user ID is also a wildcard and removes the other rules. Even in this worst case, the rule becomes the same as one used at edge firewalls. In such a way, xFilter can prevent the explosion of the number of generated rules.

When a new rule is generated, xFilter alerts the user of a target VM to solve a problem.

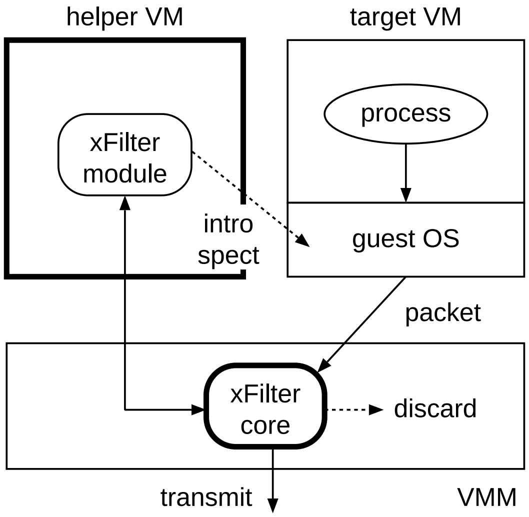


Figure 3. xFilter in the development phase.

Since the user can know which process or user is compromised, it is easy to identify compromised applications. After the user recovers that VM by fixing vulnerabilities, IaaS administrators remove the added rule. To ease their burden, it is useful to provide the interface to centrally maintain rules stored in many hosts. Furthermore, it may be possible to allow IaaS users themselves to remove rules in order not to involve IaaS administrators. If users remove rules without solving problems, xFilter would add the same rule again.

Development Support

One disadvantage of running xFilter in the VMM is the difficulty of the development of xFilter. xFilter has to be extended so that it can detect new outgoing attacks and filter packets with more information inside guest operating systems. When the developers extend xFilter, they have to modify the VMM. This causes two problems. First, the developers have to reboot the whole system to activate new xFilter whenever they modify it a bit. Second, bugs in xFilter can crash the whole VMM easily and result in rebooting the whole system. At the initial stage of the development, new implementation of xFilter includes many bugs due to the complexity of programming for VM introspection. Frequent reboots of the whole system lower the efficiency of the development.

For the purpose of debugging, xFilter allows most of its functionality to run in another VM, named the *helper VM*, as in Figure 3. When the xFilter core in the VMM intercepts a packet, it invokes an xFilter module in the helper VM. The module introspects the target VM using the function of the VMM. This architecture is similar to VMwall (Srivastava & Giffin, 2008). Since an xFilter module runs as a process in the helper VM, activating a new module only needs to restart the process. Even when a module crashes, the developers can restart the process. The crash of a module does not affect the rest of the system.

After the developers finish debugging a new xFilter module, they can embed it into the VMM without any modification. xFilter provides the common application programming interface (API) for its module both in the VMM and in the helper VM. The differences that come from where the module runs are hidden by the API provided by xFilter. For example, the method for accessing the memory of VMs is different between the VMM and the helper VM, but xFilter provides the same interface to do that. Embedding the module into the VMM is necessary in terms of performance.

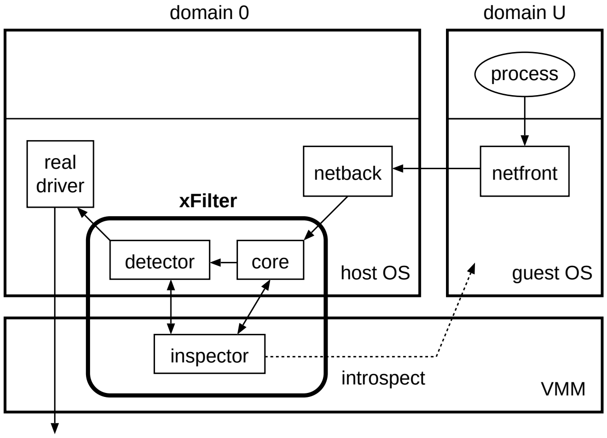


Figure 4. The architecture of xFilter in Xen.

Limitation

xFilter cannot perform pinpoint active responses 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 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 responses, 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.

IMPLEMENTATION

We have implemented xFilter in Xen 3.4.2 (Barham et al., 2003). 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. However, it is not difficult to implement xFilter in fully-virtualized Linux of Xen and KVM (Kivity & Tosatti, 2007).

To reduce the overhead of VM introspection and make packet filtering with sender information efficient, we have developed several optimization techniques. Our techniques are (1) VM introspection in the VMM, not in domain 0, (2) optimized sender traversal, (3) the decision cache, and (4) two-phase attack detection.

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 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 domain 0 and the *netback* driver invokes the *xFilter core*. If the core decides to deny 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.

When the xFilter core and detector need VM introspection, they invoke the *xFilter inspector* in the VMM by issuing a new hypervisor call. The inspector pauses domain U and introspects its

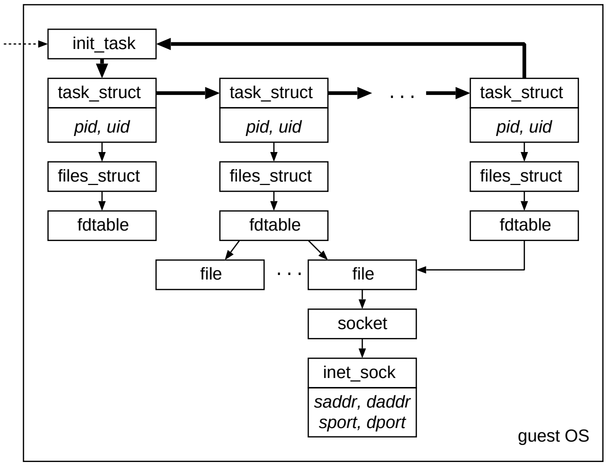


Figure 5. The traversal of kernel data structures.

memory to identify the packet sender. When it is invoked by the core, it also makes a decision on packet filtering with the obtained sender information and returns the decision to the core. When the inspector is invoked by the detector, it simply returns the sender information.

xFilter runs only the inspector in the VMM for 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.

Sender Traversal with VM Introspection

The xFilter inspector obtains information on the data structures and global symbols in guest operating system kernels from their debug information. Such debug information is stored in the DWARF format (DWARF Debugging Information Format Committee, 2010). Then the inspector translates obtained virtual addresses to physical addresses. It looks up page tables in the VMs and performs the translation by itself. It has to access several memory pages per translation, but this overhead is minimized thanks to running the inspector in the VMM. There are libraries for introspecting guest operating systems, such as XenAccess (Payne et al., 2007) and LibVMI (Payne & Leinhos, 2011). However, we could not use them because they intend to be used by user-level processes on domain 0.

Figure 5 illustrates how the xFilter inspector traverses kernel data structures. To find a process sending a particular packet, the inspector traverses the process list in domain U from from the init_task symbol. The process list consists of all task_struct objects, which contain process information such as process IDs and owner's user ID. While traversing the process list, the inspector deeply inspects the socket list that each process owns. If the inspector finds the inet_sock object for a socket whose source and destination IP addresses and port numbers match the target packet, it regards the process owning that object as a sender. When one socket is shared between multiple processes by spawning the process that created the socket, the inspector regards the original process as a sender. This approach is the same as that in iptables (Netfilter Core Team, 2001).

To introspect the guest operating system consistently, the xFilter inspector first checks spin locks for introspected data structures. For example, if the guest operating system does not acquire the spin lock for the process list, the inspector can traverse the list safely. Otherwise, the inspector aborts VM introspection and attempts to handle that packet after a while.

Optimized Sender Traversal

The xFilter inspector optimizes the above sender traversal when it decides whether a packet matches one of the filtering rules or not. Since the above algorithm needs to traverse all the processes and sockets in the guest operating system, the time is proportional to those numbers. To reduce the number of kernel objects to be introspected, the inspector first checks whether the ID or owner of each process matches one of the filtering rules while it traverses the process list. If both do not match any rules, the inspector can skip the deep traversal of the socket list owned by the process. Only for a process whose ID or owner matches at least one of the rules, the inspector examines sockets that the process opens.

Decision Cache

The xFilter core maintains *decision cache* to store 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. 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 on the basis of the TCP control bits in packet headers. When a new connection is being established and 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 entry includes source and destination IP addresses, port numbers, and *allow* or *deny* as a decision. When an existing connection is terminated or reset and the core receives a packet with the FIN or RST flag set, it removes the corresponding entry. For the other packets, the core looks up the decision cache. The core manages those entries in a least-recently-used (LRU) manner and simply invokes the inspector again if necessary entries are evicted.

Two-phase Attack Detection

Since xFilter detects attacks using sender information as well as packet headers, it must perform VM introspection for all packets even if any attacks are not mounted. The above optimization techniques such as optimized sender traversal and the decision cache cannot be used to mitigate the overhead of VM introspection in attack detection. Since there are any entries in the decision cache yet, xFilter has to traverse all processes and sockets sequentially until it find the socket sending a target packet.

To reduce the overhead under no attacks, the xFilter detector has two phases: detection and inspection. In the detection phase, the detector examines outgoing packets with only information included in packet headers. Since the detector in this phase is the same as the ones for edge firewalls, the overhead of the attack detection is minimum.

Once the detector detects an attack, it changes into the inspection phase. In this phase, whenever the detector receives a packet, it identifies the sender process from the packet information by using VM introspection. When the detector detects an attack again, it generates a *deny* rule, as shown in Figure 2. For multi-packet attacks such as portscans and DoS attacks, the detector is in the inspection phase until it detects an attack again. Note that, for single-packet attacks, the detector changes back into the detection phase soon after it inspects one packet detected in the detection phase and generates a rule.

Although two-phase attack detection can reduce detection overhead under no attacks dramatically, one drawback is that the time needed for the completion of attack detection becomes twice for multi-packet attacks. The xFilter detector has to detect two attacks in the detection and

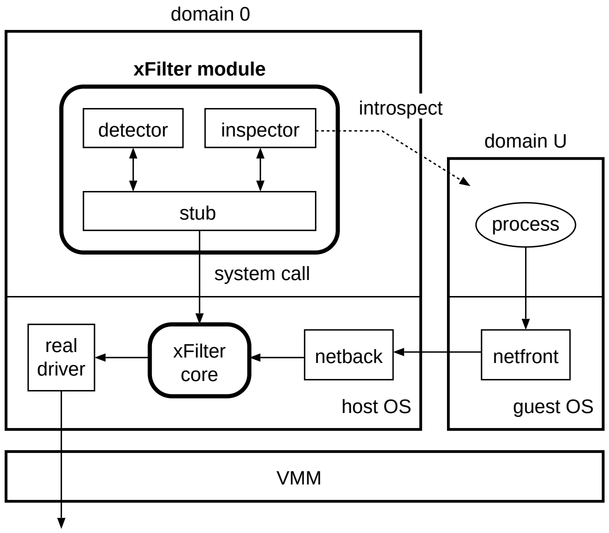


Figure 6. The architecture of xFilter in the development phase.

inspection phases, respectively. In other words, xFilter cannot stop outgoing attacks for a longer time. This may lead to false negatives because the detector cannot generate any rules unless attacks are mounted for a sufficient period. The mitigation of this problem is that cloud providers alert the VMs' users when attacks are detected in the detection phase but not detected in the inspection phase.

Support for Raw Sockets

Raw sockets are used for mounting special attacks, which cannot be mounted with regular sockets. For example, SYN flood attacks need raw sockets to send only SYN packets used for establishing TCP connections. Raw sockets enable the user to freely assemble packets with protocol headers. This means that kernel objects for raw sockets do not contain protocol information such as IP addresses and port numbers. Therefore, the xFilter inspector cannot find `inet_sock` objects sending such packets by sender traversal with VM introspection.

To find possible senders even when raw sockets are used, the xFilter inspector regards all the processes that open any raw sockets as senders. If it finds a `raw_sock` object for a raw socket during sender traversal, it records a process that owns the socket as a sender candidate. If it cannot find an appropriate `inet_sock` object after traversing all sockets, it guesses that a raw socket is probably used for sending the packet. Once xFilter detects attacks using a raw socket in this way, the inspector disables optimized sender traversal in packet filtering. It has to always traverse all sockets to ensure that target packets are not sent by legitimate processes.

xFilter Module

In the development phase, the xFilter module runs as a process in domain 0, as in Figure 6. The stub attached to the module issues a system call and waits until the xFilter core receives a packet from the netback driver. On receipt of a packet, the xFilter core wakes up the module and the system call returns the ID of the domain U sending the packet and the packet header. Then the stub pauses the domain U with the function of the VMM and invokes the module. When the module in domain 0 analyzes the memory of the domain U, it accesses the memory indirectly using the function of the VMM because domain 0 is isolated from domain U. Finally, the stub passes the decision to the xFilter core by issuing the system call again.

The cost of memory accesses of domain U from domain 0 is very high. The xFilter module has to use the hypervisor call that maps the memory pages of domain U on the address space of domain 0. Through the mapped memory pages, they can refer to the memory of the guest operating system and analyze it. To map a memory page of domain U, the hypervisor call has to add a new page table entry to the page table in domain 0. At the same time, it has to flush TLBs. In addition, the module also has to map several memory pages used for the page tables in domain U. The module looks up the page tables for the address translation. For example, it needs to map four memory pages to traverse the four-level page table. Although the development phase does not require high performance, too large overhead causes frequent TCP timeouts and makes the test of the module difficult.

To reduce the overhead of VM introspection from domain 0, we added a new hypervisor call for enabling *copy-based VM introspection*. The hypervisor call traverses the page tables of specified domain U from the VMM. Since the VMM can directly access the memory of domain U without memory mapping, there is no extra overhead for accessing the page tables. Then, the hypervisor call copies the contents of the target memory page to the buffer passed as its argument. This memory copy needs to neither add a new page table entry to the page table of domain 0 nor flush TLBs.

In addition, to reduce the communication overhead between the xFilter core and module, the core gathers multiple packets and passes them to the module at once. Since it is inefficient to traverse the process list for each packet, the module finds multiple sender processes for all the passed packets during one traversal of the process list. We call this optimization technique *parallel sender traversal*. This can reduce the number of VM introspection in total. To avoid excessively increasing the latency of packet sending, the core waits for invoking the module only for a short period.

EXPERIMENTS

We performed experiments for demonstrating the effectiveness of xFilter and examining its overheads. As an example of the xFilter detector, we have implemented a portscan detector. 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.

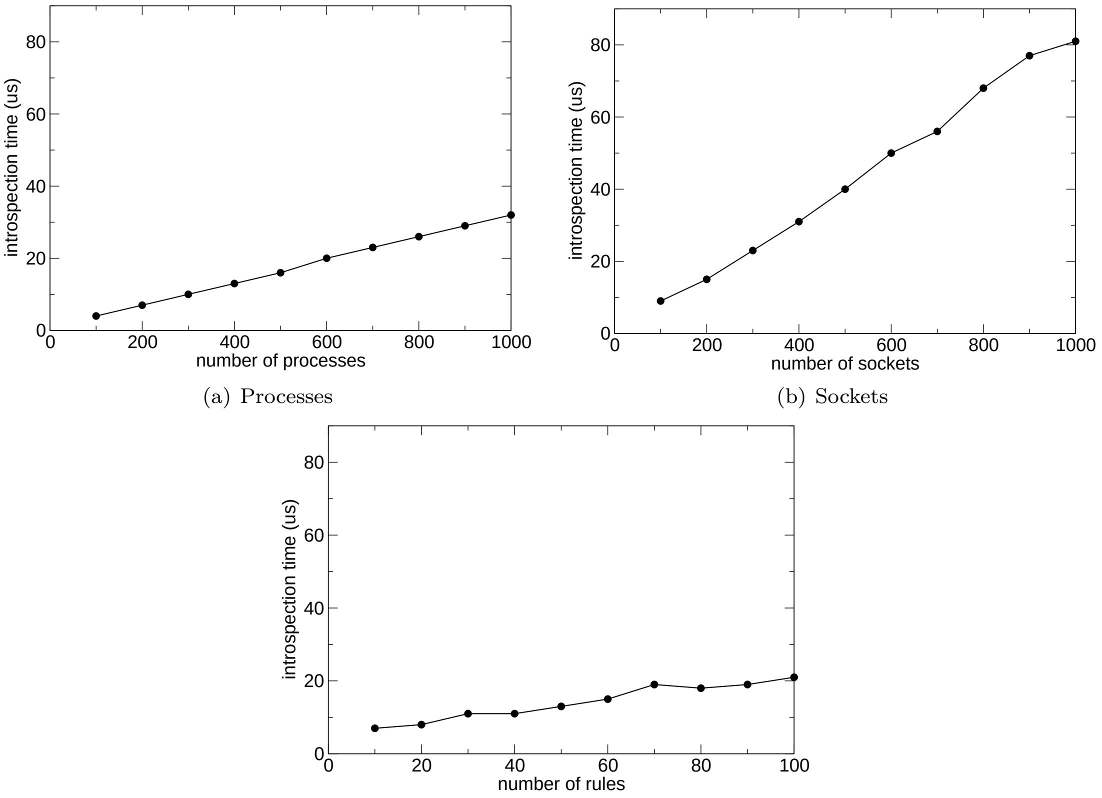
Self-protection against Portscans

To demonstrate that xFilter enables self-protection against portscans, we performed portscans from a victim VM to the outside hosts using nmap (Lyon, 1997). 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, the xFilter detector could detect the both types of portscans and stop the successive attacks by automatically generating a process-level rule like the first rule in Figure 2. 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, the detector also detected the portscans and generated process-level rules for each process. After we continued the portscans, the detector automatically merged five these rules into one user-level rule like the second rule in Figure 2 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.

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 total number of processes in a VM and measured



(a) Processes

(b) Sockets

(c) Rules

Figure 7. The introspection time for the various numbers of kernel objects and filtering rules.

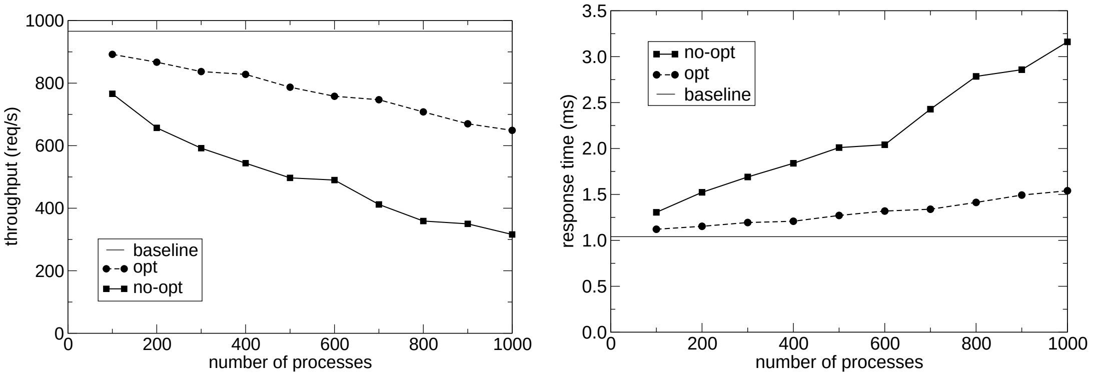
the execution time. We specified a non-existent process ID in a filtering rule so that the xFilter inspector traversed the entire process list and checked the process IDs of all the processes. In this experiment, the inspector did not deeply inspect kernel data structures for sockets. Figure 7(a) shows the execution time, which is proportional to the number of processes and takes 31 ns per process.

Second, we changed the total number of sockets created by one process 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 7(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 specified a non-existent process ID in all the rules. Figure 7(c) 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.

Effect of Optimized Sender Traversal

To examine the effects of various optimization techniques for mitigating the overheads of VM introspection, we measured the throughput and the response time of the Apache web server (Apache Software Foundation, 1995). 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, which was relatively small. The baseline



(a) Throughput

(b) Response time

Figure 8. The performance improvement by optimized sender traversal.

is the performance when we did not use xFilter. At that time, the throughput was 966 requests/s and the response time was 1.04 ms.

First, we examined the effect of optimized sender traversal. For this purpose, we changed the number of processes and measured the web performance. We used the same filtering rule as that for processes in the previous section. Figure 8 shows the throughput and response time when we enabled and disabled optimized sender traversal. The web performance in both cases degraded in proportion to the number of processes. From these results, it is shown that optimized sender traversal achieved 58 % performance improvement for 500 processes. Whereas the web performance degraded only by 19 % with optimized sender traversal, it degraded by 49 % without this optimization.

Effect of the Decision Cache

To examine the effect of the decision cache in addition to optimized sender traversal, we compared the web performance when the decision cache was enabled with that when it was disabled. We conducted three experiments for various numbers of processes, sockets, and rules. For each, we used the same filtering rule as that in the above section.

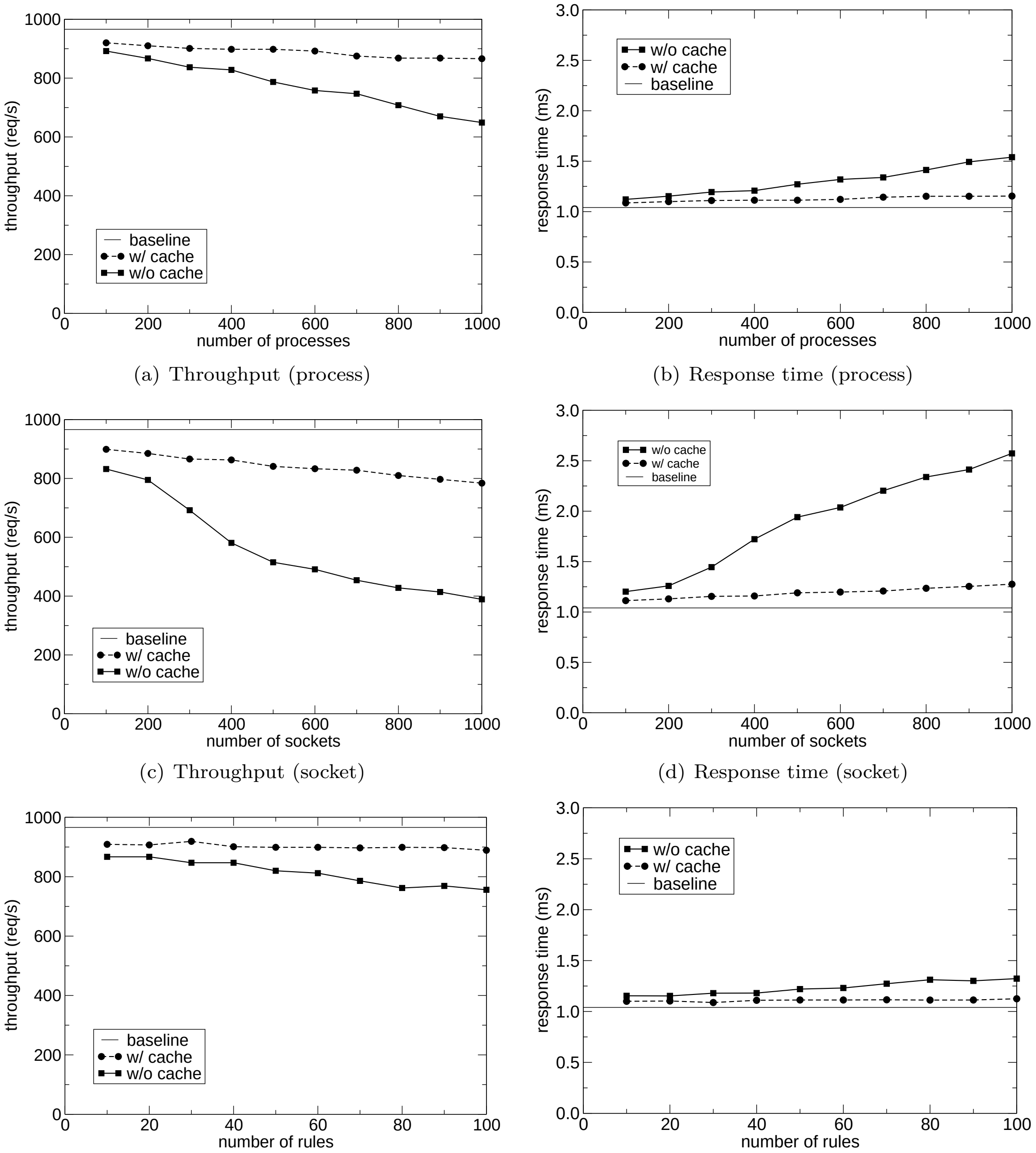
First, we measured the web performance when we changed the number of processes. Figure 9(a) and 9(b) show the throughput and response time, respectively. These figures show that the decision cache improved the web performance by 14 % for 500 processes. The web performance degraded only by 7 % with the decision cache, whereas it degraded by 19 % without the cache.

Next, we measured the web performance for various numbers of sockets. As shown in Figure 9(c) and 9(d), the performance degradation is proportional to the number of sockets, which is similar to the above experiment for processes. However, the web performance was improved by 63 % for 500 sockets. This is because optimized sender traversal cannot reduce the number of sockets to be introspected when their owner process should be checked. While the performance degradation was only 13 % with the decision cache, it was 47 % without the cache. 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. Figure 9(e) and 9(f) show the results. The performance improvement by the decision cache was 17 % even for 100 rules. In reality, if 100 rules are needed, the system would be completely compromised.

Effect of Two-phase Attack Detection

To examine the effect of two-phase attack detection, we measured the performance of the web server when xFilter had no filtering rules yet. In this experiment, the xFilter detector performed



(a) Throughput (process)

(b) Response time (process)

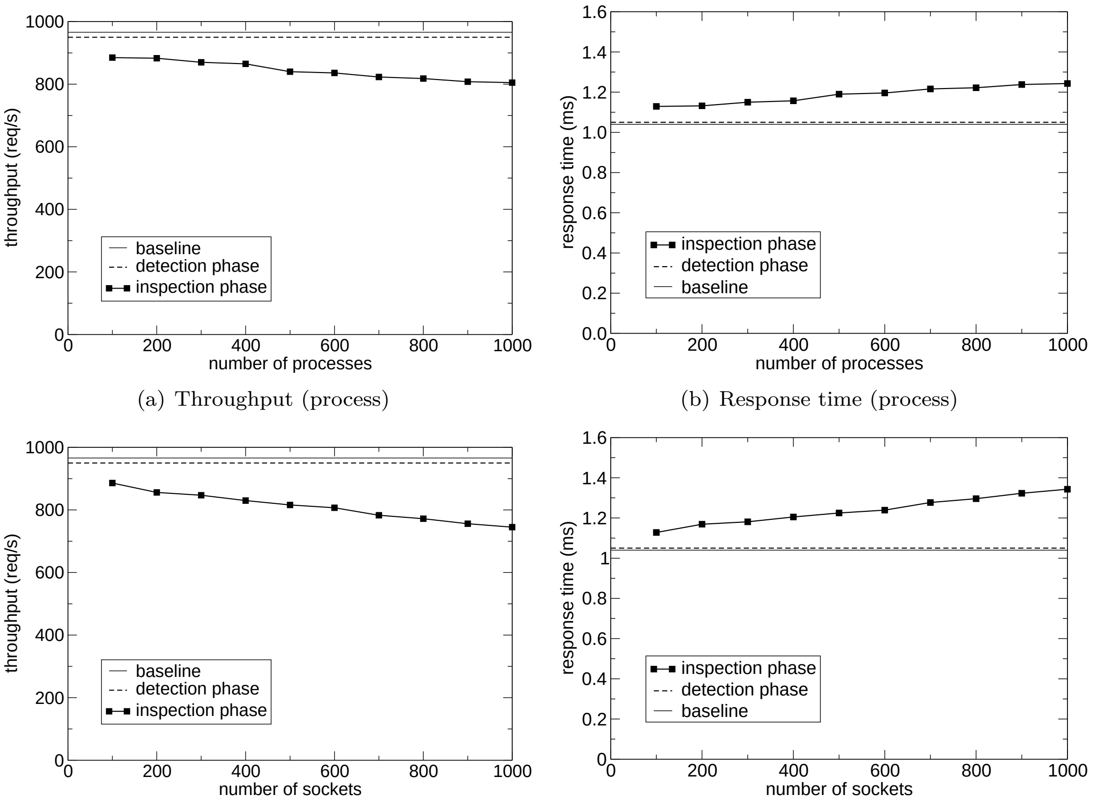
(c) Throughput (socket)

(d) Response time (socket)

(e) Throughput (rule)

(f) Response time (rule)

Figure 9. The performance improvement by the decision cache.



(a) Throughput (process)

(b) Response time (process)

(c) Throughput (socket)

(d) Response time (socket)

Figure 10. The performance degradation by attack detection.

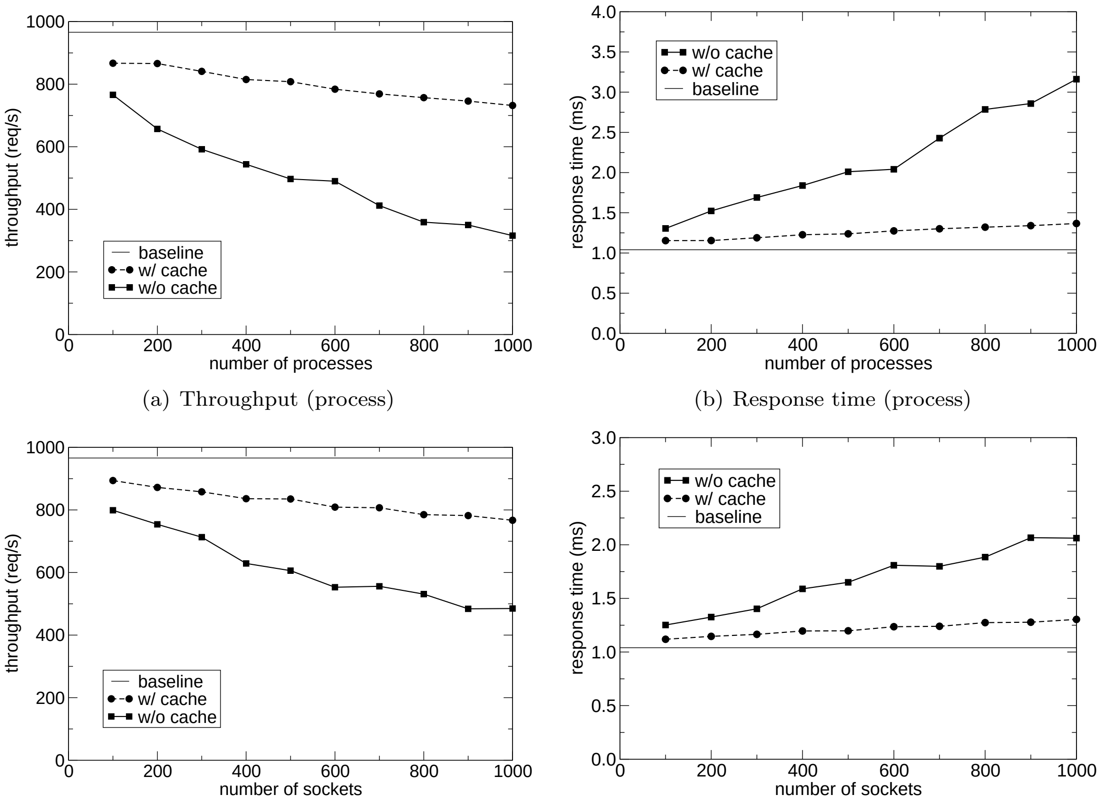
the portscan detection for every packet. Figure 10 shows the web performance when we changed the number of processes and sockets.

In the detection phase, the performance degradation was only 1 % because the detector did not perform VM introspection. This is the overhead for detecting portscans only from packet headers. In the inspection phase, on the other hand, the performance degraded as the numbers of processes and sockets increased. The performance degraded by 13 % for 500 processes, while it degraded by 16 % for 500 sockets. These results mean that the overhead of the detector is small enough until outgoing attacks are detected.

Overheads for Supporting Raw Sockets

To examine the filtering performance after xFilter detected an attack using raw sockets, we measured the web performance for various numbers of processes and sockets. When attacks are mounted using raw sockets, the xFilter inspector has to traverse all processes and sockets and then cannot use optimized sender traversal. We measured the performance with and without the decision cache because the decision cache is useful even for attacks using raw sockets. The decision cache can check packet headers assembled with raw sockets. We used the same filtering rules as those in the above section.

Figure 11(a) and 11(b) show the web performance when we changed the number of processes. Without the decision cache, the performance degraded by 49 % for 500 processes, compared with the baseline. The decision cache improved the performance by 62 %, but the performance was still 10 % lower than that in packet filtering for regular sockets. Similarly, Figure 11(c) and 11(d) show



(a) Throughput (process)

(b) Response time (process)

(c) Throughput (socket)

(d) Response time (socket)

Figure 11. The web performance under attacks with raw sockets.

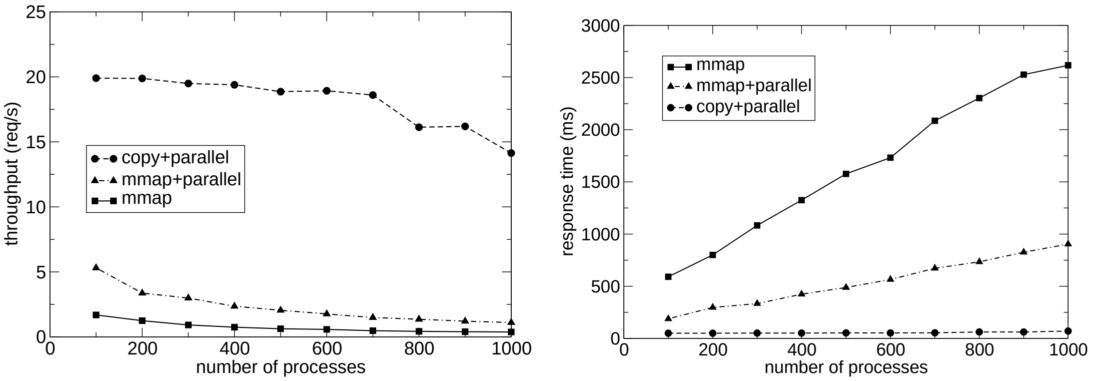
the performance when we changed the number of sockets. For 500 sockets, the performance without the cache was 38 % lower than the baseline. Thanks to the cache, the performance was improved by 37 %. The overhead for dealing with raw sockets was 8 %.

Performance in the Development Phase

To examine the web performance in the development phase, we ran our xFilter module in the helper VM and measured the throughput and response time. The module performed packet filtering with sender information and used optimized sender traversal and the decision cache. In addition, we examined the effect of the optimization techniques specific for the development phase: copy-based VM introspection and parallel sender traversal. For comparison, we also measured the web performance when the module used traditional mmap-based VM introspection.

Figure 12 shows the results. For 500 processes, the optimization of parallel sender traversal obtained 3.2 times higher performance, compared with per-packet sender traversal. When the module performed copy-based VM introspection as well, the performance became 9.2 times higher than mmap-based one. However, the performance was only 2 % of that in xFilter running in the VMM. This means that running xFilter in the VMM is indispensable for the production phase.

Although the architecture of VMwall (Srivastava & Giffin, 2008) is similar to ours in the development phase, it is reported that the overhead of VMwall was 7 % at maximum. The primary reason is that the overhead of VMwall was measured while a large file of 175 MB was transferred. Since VMwall uses a mechanism similar to our decision cache, it did not perform VM introspection for most of the packets. In our experiment, xFilter performed VM introspection more frequently



(a) Throughput

(b) Response time

Figure 12. The web performance in the development phase.

because of transferring a much smaller file of 50 KB. The second reason is that the overhead of VMwall may be measured in small numbers of processes and sockets. However, these numbers are not shown in the literature.

CONCLUSION

In this paper, we proposed an efficient and fine-grained VMM-level packet filter, called xFilter, for self-protection of IaaS clouds. xFilter uses VM introspection to obtain information on sender processes in VMs. To make packet filtering with VM introspection efficient, we introduced four optimization techniques: VM introspection in the VMM, optimized sender traversal, the decision cache, and two-phase attack detection. Our experiments showed that, thanks to these techniques, the performance degradation due to xFilter was usually less than 16 %. Under no attacks, the overhead was only 1 %.

One of our future work is using other information on sender processes for packet filtering. For example, grouping processes with information on their ancestors may be helpful. Also, it is necessary to support VM migration. To continue to apply filtering rules after VMs are migrated to other hosts, we has to migrate rules together with VMs. Another direction is considering hardware-level I/O virtualization such as SR-IOV (PCI-SIG, 2010). Since the VMM cannot capture packets in SR-IOV, we are planning to combine packet filtering at edge firewalls and VM introspection at hosts running VMs.

ACKNOWLEDGMENT

This research was supported in part by JST, CREST.

References

Amazon, Inc. (2006). *Amazon Elastic Compute Cloud.* `http://aws.amazon.com/ec2/`.

Amazon, Inc. (2009). *Amazon Web Services: Overview of Security Processes.* `http://aws.amazon.com/security/`.

Apache Software Foundation. (1995). *Apache HTTP Server Project.* `http://httpd.apache.org/`.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., ... Warfield, A. (2003). Xen and the Art of Virtualization. In *Proceedings of Symposium on Operating Systems Principles* (pp. 164–177).

Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). Ether: Malware analysis via Hardware Virtualization Extensions. In *Proceedings of Conference on Computer and Communications Security* (pp. 51–62).

DWARF Debugging Information Format Committee. (2010). *DWARF Debugging Information Format Version 4.* `http://dwarfstd.org/`.

Garfinkel, T., & Rosenblum, M. (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium* (pp. 191–206).

Garg, S., & Saran, H. (2008). Anti-DDoS Virtualized Operating System. In *Proceedings of International Conference on Availability, Reliability and Security* (pp. 667–674).

Jiang, X., Wang, X., & Xu, D. (2007). Stealthy Malware Detection Through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction. In *Proceedings of Conference on Computer and Communications Security* (pp. 128–138).

Johns, M. (1993). *Identification Protocol.* RFC 1413.

Joshi, A., King, S., Dunlap, G., & Chen, P. (2005). Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In *Proceedings of Symposium Operating Systems Principles* (pp. 91–104).

Kivity, A., & Tosatti, M. (2007). *Kernel Based Virtual Machine.* `http://www.linux-kvm.org/`.

Kourai, K., Azumi, T., & Chiba, S. (2012). A Self-protection Mechanism against Stepping-stone Attacks for IaaS Clouds. In *Proceedings of International Conference on Autonomic and Trusted Computing* (pp. 539–546).

Kourai, K., Chiba, S., & Masuda, T. (1998). Operating System Support for Easy Development of Distributed File Systems. In *Proceedings of International Conference on Parallel and Distributed Computing and Systems* (pp. 551–554).

Lyon, G. (1997). *Nmap – Free Security Scanner For Network Exploration & Security Audits.* `http://nmap.org/`.

Netfilter Core Team. (2001). *The netfilter.org Project.* http://www.netfilter.org/.

Payne, B., Carbone, M., & Lee, W. (2007). Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of Annual Conference on Computer Security Applications* (pp. 385–397).

Payne, B., Carbone, M., Sharif, M., & Lee, W. (2008). Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of Symposium on Security and Privacy* (pp. 233–247).

Payne, B., & Leinhos, M. (2011). *LibVMI.* `http://code.google.com/p/vmitools/`.

PCI-SIG. (2010). *Single Root I/O Virtualization and Sharing 1.1 Specification.* `http://www.pcisig.com/`.

Petroni, Jr., N., & Hicks, M. (2007). Automated Detection of Persistent Kernel Control-flow Attacks. In *Proceedings of Conference on Computer and Communications Security* (pp. 103–115).

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., ... Neuhauser, W. (1992). Overview of the Chorus Distributed Operating System. In *Proceedings of Symposium on Microkernels and Other Kernel Architectures* (pp. 39–69).

Srivastava, A., & Giffin, J. (2008). Tamper-resistant, Application-aware Blocking of Malicious Network Connections. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection* (pp. 39–58).

Staniford-Chen, S., & Heberlein, L. (1995). Holding Intruders Accountable on the Internet. In *Proceedings of Symposium on Security and Privacy* (pp. 39–49).