Novel Intrusion Detection System for Cloud Computing: A Case Study

Ming-Yi Liao[1(✉)], Zhi-Kai Mo[1], Mon-Yen Luo[2], Chu-Sing Yang[1], and Jiann-Liang Chen[3]

[1] Department of Electrical Engineering, Institute of Computer and Communication Engineering, National Cheng Kung University, Tainan, Taiwan, ROC
myliao@ee.ncku.edu.tw

[2] Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, Kaohsiung, Taiwan, ROC

[3] Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, ROC

Abstract. Because of the growth in cloud computing and manturity of virtualization technology, many enterprises are virtualizing their servers to increase server utilization and lower costs. However, the complex network topology arising from virtualization makes clouds vulnerable, and security breaches have occurred on cloud computing platforms in recent years. Therefore, a comprehensive mechanism for detecting and preventing malicious traffic is necessary. We propose a network intrusion detection system that is based on a virtualization platform. This system, developed from a multipattern based network traffic classifier, collects packets from the virtual network environment and analyzes their content by using deep packet inspection for identifying malicious network traffic and intrusion attempts. We improve the intrusion detection features of the network traffic classifier and deploy it on a Xen virtualization platform. Our system can be combined with the Linux Netfilter framework to monitor inter-virtual-machine communications in the virtualization platform. It efficiently inspects packets and instantly protects the cloud computing environment from malicious traffic.

Keywords: Cloud computing · Deep packet inspection · Intrusion detection system

1 Introduction

With the rise of cloud computing and maturity of virtualization technology, many enterprises are virtualizing their servers to enhance server utilization and reduce costs. However, the security events that have occurred on cloud computing platforms in recent years not only cause economic loss but also affect user privacy. Potential security risks in virtualized environments include disclosure of server configuration, disclosure of virtual machine (VM) information by a VM manager, and using VMs as bouncers for attacking other targets. To secure cloud computing networks, a comprehensive mechanism that detects and prevents malicious traffic is essential. A critical task in cloud security is providing virtual host, and hypervisor protection, anti-malware and intrusion prevention solution. Inter host communication among VMs cannot be monitored using security systems outside the host. Some solutions entail establishing an agent in every guest VM, causing redundant overhead because all agents run simultaneously. To secure cloud computing networks, we propose a network intrusion detection system (IDS) that is based on virtualization platforms. Our system can be combined with the Netfilter framework [1] for efficiently inspecting packets and instantly protecting the cloud computing environment from malicious traffic. The remainder of this paper is organized as follows. In Sect. 2, we introduce the study background and related research. Section 3 describes the implementation of our system architecture in an IDS. Section 4 describes the deployment of the IDS in a Xen virtualization platform. Section 5 describes our experiments and discusses the evaluation results, and Sect. 6 presents the conclusions.

2 Related Works

Libpcap [2] is a widely used system-independent interface for user-level packet capture that provides a portable framework for low-level network monitoring. Some IDSs and packet classifiers, such as Snort [3] and Bro [4], use libpcap. Another approach is to capture packets through a communication application program interface between the kernel and user-space in Linux [5].

Port-based classification is traditionally used in packet classification [6]. Statistics-based classification is an approach that indirectly analyzes packets [7, 8]. This method performs traffic classification without accessing sensitive data. Usually, statistics-based methods provide features of network flow for machine learning and heuristic algorithms [9]. Each feature can be considered a dimension of a vector space [10]. Data clustering algorithms are applied for data training and modeling. Deep Packet Inspection (DPI) technique matchs a predefined signature for traffic classification and intrusion detection [11]. Regular expression engines process a regular expression statement by constructing a deterministic finite automaton (DFA) [12, 13] with a time and memory cost of $O(2^m)$ for a regular expression of size m running DFA on a string of size n in time O(n). In protocol decoding, packet payloads are parsed according to the header of the application layer [14], and connections are tracked using finite state automata. The classifier [15] is a DPI system based on the Linux Netfilter framework.

Snort is an open source network IDS that detects packets through signature-based methods. Bro is a passive, open source network traffic analyzer. It supports many application-layer protocols, including DNS, FTP, HTTP, IRC, SMTP, SSH, and SSL.

Some open source platforms, and commercial solutions use virtualized servers for cloud computing. The Kernel-based Virtual Machine (KVM) [16] is an open source VM built in the Linux kernel. The KVM supports Intel VT and AMD-V virtualization, and combines the hypervisor and operating system kernel for reduced redundancy and enhanced efficiency. Xen [17] is an open source VM developed by the University of Cambridge, and runs at most 128 complete operating systems in a host.

Restricting user access to the critical resources in a virtual environment [18] is one facilitative approach to prevent the network in cloud computing from malicious threats. Another approach involves binding users to different security groups according to their anomaly levels [19]. To prevent sniffing and spoofing in virtual networks, a network model that isolates a group of VMs proposed [20]. VMs are split into shared subnetworks and are assigned unique IDs, and these IDs prevent communication between subnetworks. In addition, firewall is established between the subnetworks to prevent spoofing. A VM monitor based IPS [21] is proposed for detecting intrusion attempts in the Xen virtualization platform.

3 IDS Implementation

3.1 Multipattern Based Packet Classifier Workflow

The classifier is connected to the Netfilter and processes packets through the PROMISC hook, using connection tracking for efficient packet inspection. When packets pass through the classifier module, a sequence of inspections is triggered. First, whether each packet is an IP packet is determined. Subsequently, the classifier searches connection tracking data to ensure that the connection associated with this packet was identified previously. If the connection is unidentified, the classifier parses the packet payload and begin pattern matching. The classifier queries the rules in two steps because variable pattern matching is more time intensive than matching processes. If no match is identified after fixed and arithmetic pattern matching, variable pattern matching is applied. Finally, the classifier matches the rule again. If the matching is successful, information about the application type is written to the connection tracking data. However, when the classifier matches the first packet in a connection, connection tracking of all subsequent packets in the connection is ignored. Therefore, some exploitative behaviors and malicious connections appearing in the general protocol cannot be identified. Hence, we propose an improvement to this classifier, that entails modifying the packet matching procedure.

3.2 Multipattern Based Classifier with State Checking Function

As discussed in Sect. 3.1, connection tracking ignores certain exploitative and malicious connections in the general protocol. The Heartbleed exploit [24], a serious vulnerability in the widely used OpenSSL cryptographic software library, illustrates the vulnerability of connection tracking. It was discovered by a team of security engineers at Codenomicon and Neel Mehta of Google Security. This vulnerability arising from an implementation problem in some versions of the OpenSSL library, enables anyone on the Internet to read system memory. Attackers could steal user passwords and secret keys used to encrypt the traffic. Heartbleed exploits the fact that the OpenSSL software does not check the data length in the heartbeat protocol (RFC 6520), which is used to negotiate and monitor service availability. The heartbeat protocol includes message payload and payload length, and a heartbeat request is sent to obtain a corresponding response message. The receiver responds with a message that carries a copy of the requested payload. When an attacker sends a request where the payload length is greater than the actual length, the vulnerable OpenSSL server responds with data of the length specified by the request, causing a memory leak.

The heartbeat protocol is an extension of TLS/SSL, and the exploit occurs after a TLS connection is established. The conventional classifier detects the TLS connection by matching the hello packet but ignores the subsequent packets; therefore, it cannot detect the Heartbleed exploit. Another example is CVE-2014-3466 [25], which occurs in GnuTLS, a secure communications library that implements the SSL and TLS protocols. GnuTLS does not check the length of the session ID in the ServerHello message; this vulnerability enables attackers to overflow stack buffer and execute arbitrary code, which cannot be detected by the conventional classifier. We modified the packet matching procedure and added functions for checking the packet state and format. A sequence of functions is designed for detecting exploits and malware. Of the four signature matching patterns, arithmetic matching is used to calculate the data length and determine whether the length equals the value of the specific bytes; thus, it is similar to protocol decoding. We extend this functionality to intrusion detection, for detecting the Heartbleed exploit and botnets. If all packets are processed, the processing costs increase. Therefore, each function is assigned to a corresponding protocol. An easy solution is to construct a list of functions and search specific functions according to the application protocol of the packets; however, searching entails more costs.

We use connection tracking for assigning state-checking functions. When a packet is identified, information about the service type and matched rule is returned from the rule set, and connection tracking writes this information to the corresponding hash table. Therefore, we set a pointer in the rule entry to the corresponding functions so that packets with identified service types can be processed through predefined state-checking functions. Thus, packets are checked according to a specific protocol, reducing packet processing costs. Figure 1 illustrates the procedure for assigning state-checking functions by using connection tracking; HTTP and TLS are two example protocols.

The work flow of the classifier is in Fig. 2. When a packet enters the classifier, the classifier searches the corresponding entry in the connection tracking table. If this connection was not tracked, the classifier performs signature matching to identify the packet. If the classifier recognizes the packet service type, it writes the information to

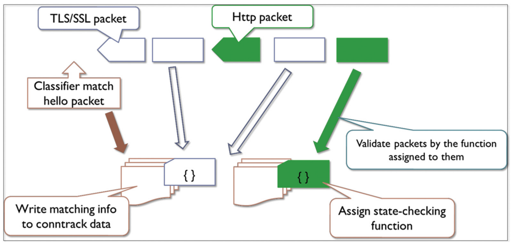

TLS/SSL packet
Http packet
Classifier match hello packet
Validate packets by the function assigned to them
{ }
{ }
Write matching info to conntrack data
Assign state-checking function

Fig. 1. Assigning state-checking functions to specific packets

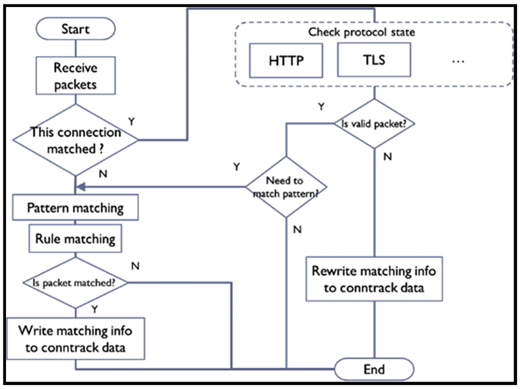


Fig. 2. Flow diagram of the classifier with state-checking function

the connection tracking entry. The subsequent packets belonging to the connection are processed using state-checking functions for protocol validation. Malicious traffic appearing in the general protocol must be detected using string matching; therefore, the state-checking function should determine packet types that match patterns. For example, HTTP botnets request pages and send commands using HTTP request; the state-checking function of HTTP identifies the request packets and forwards them for pattern matching. If a packet in a flow is considered malicious, the classifier rewrites the connection tracking data of the flow and assigns another state-checking function to it. The other packets in the flow are processed using state-checking function of HTTP. This approach comprehensively detects intrusions.

Here, we analyze the running time of our modified classifier. We first calculate the running time of packet matching without connection tracking. We assume that there are n packets in a flow. The running time of arithmetic pattern matching is t_a, and t_p is the sum of running times for fixed and variable pattern matching and rule matching. The running time for the state-checking function is t_s. The time for matching all packets is obtained as follows.

$$T_{all} = n(t_a + t_p).$$

With connection tracking, only m packets must be matched by the classifier. The time consumed is represented as follows.

$$T_{ct} = m(t_a + t_p).$$

We add the mechanism of the state-checking function to connection tracking and increase the running time ($T'_{ct'}$), which is represented as follows.

$$\begin{aligned} T'_{ct} &= m(t_p + t_a) + t_s(n - m) \\ &= \mathrm{n}t_s + (t_{p=} + t_a - t_s). \end{aligned}$$

Because the state-checking function and arithmetic pattern matching employ the same methods for checking packets, we assume that they consume the same amount of time. Practically, the state-checking function for a specific application is a subset of the arithmetic patterns; therefore, the running time of the state-checking function is less than that of arithmetic pattern matching. If we assume $t_s = t_a$, we can rewrite T'_{ct}:

$$T'_{ct} = \mathrm{n}t_a + mt_p.$$

The running time of the classifier with the state-checking function is less than that for matching all packets. We calculate their time consumption ratio:

$$\frac{T'_{ct}}{T_{all}} = \frac{nt_a + mt_p}{nt_a + nt_p}.$$

In related research, t_p is approximately 66.9 ms and t_a is approximately 5.3 ms in the worst case. For example, if there are ten packets and the first packet is matched using string pattern matching, the ratio is 0.166, which is 66 % higher than that of the conventional classifier; nevertheless, the time consumption of the modified classifier is much lower than that for matching all packets. In the subsequent section, we prove that the proposed design is reasonable in combination with the state-checking function and connection tracking.

4 Deployment

We deploy the classifier on a Xen platform and test the proposed design in several network scenarios to demostrate its suitability in cloud computing environments. The classifier is a kernel module connected to the Netfilter and processes packets from and to the VM detecting intrusions; however, comprehensively securing virtual networks in virtualization platforms is not easy. Networking in Xen can be constructed using the Linux bridge or Open vSwitch in Domain 0. We set up the classifier in combination with the virtual switch or the bridge, as explained subsequently.

4.1 Open vSwitch

Open vSwitch is a multilayer virtual switch designed to enable effective network automation through programmatic extensions. If the Xen network is connected by a virtual switch, the function of the bridge module in Domain 0 is replaced. All packets transmited to and from VMs are forwarded by the virtual switch; therefore, the Netfilter does not receive these packets, and the classifier must run in the offline mode. We enable the mirror port to collect traffic from all VMs and create a virtual interface for receiving packets mirrored from the virtual switch. Figure 3 shows the architecture of the virtual network with Open vSwitch. If more than one physical machine is used, the virtual switches on each host mirror its ports to a central traffic detection system installed in the classifier, as shown in Fig. 4.

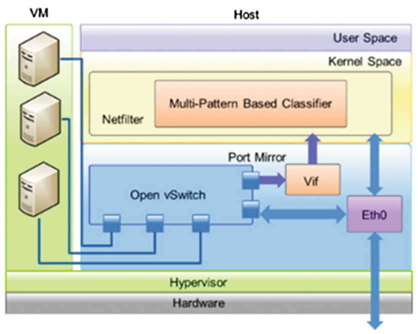


Fig. 3. Multipattern based classifier with Open vSwitch

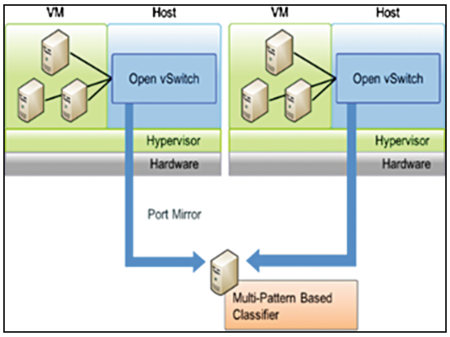


Fig. 4. Mirroring ports outside the multipattern based classifier

4.2 Linux Bridge

Linux provides virtual bridges for connecting several Ethernet segments independent of the protocol. The Linux bridge by default connects the network interface of Domain 0 and Domain Us in the Xen network. The Bridge-Netfilter is a Netfilter component that provides several modules for packet manipulation at the data link layer. In addition, it can be connected to user-defined modules. The five hooks in both layers are the same, but the BROUTING hook in the Bridge-Netfilter redirects packets between interfaces with different subnets. To filter the packets passing through the bridge, connecting the classifier to the Bridge-Netfilter is an easy approach. However, the classifier requires the connection tracking module (for enhanced packet matching efficiency), which the Bridge-Netfilter does not support. We use sysctl to modify the configuration to enable packets in the bridge to pass through the PREROUTING hook of the Netfilter and return to the Bridge-Netfilter so that they can beprocessed by the classifier, as depicted in Fig. 5.

However, the NAT table of the Netfilter is traversed only by the first packet in each connection. Previous approaches overcome this problem by defining a PROMISC hook

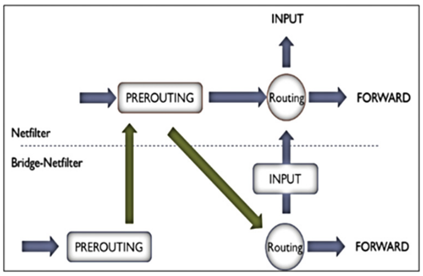


Fig. 5. PREROUTING hook of the Netfilter is enabled in the bridge

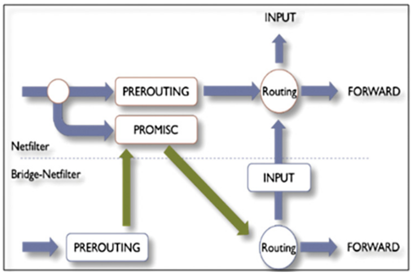


Fig. 6. The entrance of PREROUTING is substituted with that of PROMISC

in the Netfilter. We substitute the entrance of the PREROUTING hook with that of PROMISC. Figure 6 shows the path modified for the classifier in the Bridge-Netfilter.

5 Evaluation

We conducted evaluation experiments to prove that our system is a suitable IDS for cloud computing environments. To test packet loss rate and bit rate, we set up the classifier on a Xen virtualization platform and operated VMs to generate packets. Moreover, we tested the accuracy of our classifier by replaying traffic samples of malware and exploits. The experimental environment is described in the subsequent sections.

5.1 Environment

Table 1 details our environment. We constructed a Xen system and installed software on VMs. The Linux kernel was rebuilt from the source to enable installing our modified classifier. The iptables are required to enable the classifier; hence, we updated the iptables for compatibility with a newer kernel. The Linux bridge and Open vSwitch were not enabled simultaneously because of compatibility concerns.

Table 1. Experimental environment

Component	Software
Dom0	CentOS 6.4, Linux kernel 3.12.4, iptables 1.4.21, Snort 2.9.6.1
DomU	CentOS 6.4
Xen	Xen 4.3.1

5.2 Packet Loss Rate and Throughput Testing

The experimental scenario is illustrated in Fig. 7. We used two VMs; one floods user data protocol (UDP) traffic to the other, which receives and counts the packets. We gathered statistics from the network interfaces of the VMs to determine the number of packets sent and received. For comparison, we individually tested the modified classifier and Snort. Snort was set in the inline mode for detecting traffic passing through the bridge. Because the classifier contains 397 rules, we enabled only 383 exploit-kit rules in Snort. We used Hping [26] to generate UDP traffic, and varied the source and destination ports of each packet from 1 to 65,535 to prevent connection tracking from ignoring most packets. We tested packet sizes ranging from 64 bytes to 1,512 bytes and observed the variation. The experimental results are presented in Figs. 8 and 9. In Fig. 8, the Snort packet loss rate fluctuates between 17 % and 36 % and is higher than that of the classifier, of which the loss rate is < 1 %. Figure 9 reveals that our classifier affords a higher bit rate than Snort does.

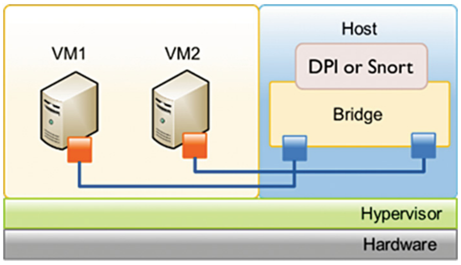


Fig. 7. Connecting VMs by using the Linux bridge

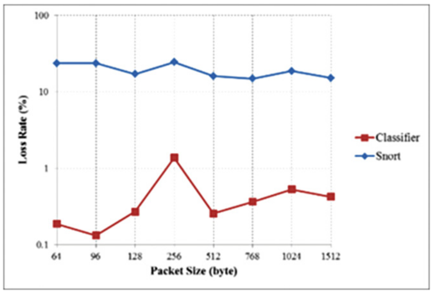


Fig. 8. Packet loss rate of traffic between VMs

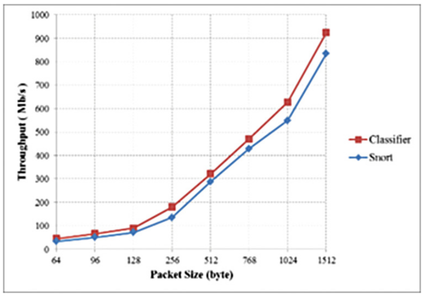


Fig. 9. Throughput of traffic between VMs

In another experiment, we flood UDP traffic from a PC outside the virtualization platform to a VM. The test environment is depicted in Fig. 10.

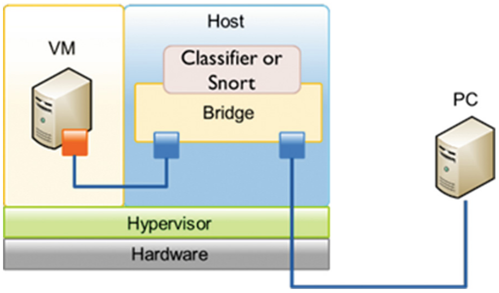


Fig. 10. Connecting a VM and a PC by using the Linux bridge

As in the earlier experiment, we conducted separate experiments with the classifier and Snort. As Fig. 11 shows, the packet loss rate of Snort decreases from 17 % to 7 %, and the packet loss rate of the classifier does not exceed 0.01 %. In Fig. 12, the bit rate of Snort is higher than that of our classifier, possibly because packets pass through the classifier at a lower speed to ensure transmission. Snort runs in the user space and

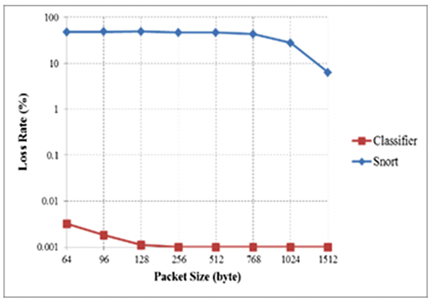


Fig. 11. Packet loss rate of traffic from a PC to a VM

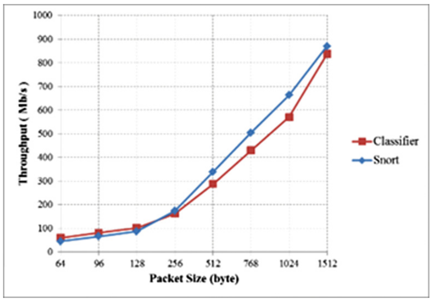


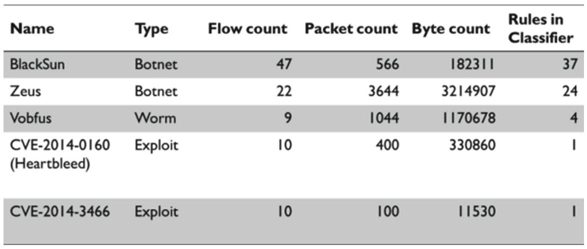
Fig. 12. Throughput of traffic from a PC to a VM

receives packets from the kernel by AF_PACKET, which causes severe packet loss. These experiments prove that our classifier remains stable during packet transmissions.

5.3 Accuracy Testing

We collected five types of malicious traffic a worm, two botnets, and two exploits to test the accuracy of the classifier. Table 2 describes the flow count, packet count, and the corresponding rules in the classifier for these samples. These samples are stored in the PCAP files and are split into smaller files according to TCP streams. We shuffled these files and replayed them to generate malicious traffic.

Table 2. Traffic samples of malware and exploits



Name	Type	Flow count	Packet count	Byte count	Rules in Classifier
BlackSun	Botnet	47	566	182311	37
Zeus	Botnet	22	3644	3214907	24
Vobfus	Worm	9	1044	1170678	4
CVE-2014-0160 (Heartbleed)	Exploit	10	400	330860	1
CVE-2014-3466	Exploit	10	100	11530	1

The accuracy test results of the conventional and modified classifiers as listed in Tables 3 and 4, respectively. Comparing these results reveals that a higher number of BlackSun and TLS exploits are detected by the modified classifier than by the conventional classifier. The conventional classifier cannot detect exploits inside TLS, such as the Heartbleed exploit and CVE-2014-3466, because they are ignored by connection tracking. A higher number of BlackSun packets are detected by the modified classifier because these packets are commands following the normal HTTP requests in a connection. The other matched packets include HTTP and FTP packets. Some packets are

Table 3. Flows matched by the classifier

Name	Flow count	Packet count	Byte count
BlackSun	37	425	128487
Zeus	21	3643	3164178
Vobfus	9	1044	1157502
CVE-2014-0160 (Heartbleed)	0	0	0
CVE-2014-3466	0	0	0
TLS	20	410	186390
http	8	110	45085
ftp	2	27	2330

Table 4. Flows matched by the modified classifier

Name	Flow count	Packet count	Byte count
BlackSun	38	451	141601
Zeus	21	3643	3164178
Vobfus	9	1044	1157502
CVE-2014-0160 (Heartbleed)	10	310	176260
CVE-2014-3466	10	100	10130
TLS	0	0	0
http	7	84	31971
ftp	2	27	2330

not shown in the results because they are malicious TCP packets that are not received by the Netfilter. The HTTP packets are requests for elements in the pages of BlackSun, and the FTP packets are file transmissions without significant signatures. Thus, we demonstrate that our modification enhances the functionality of intrusion detection.

6 Conclusion

We propose a novel IDS that is an improvement of multipattern based network packet classifier. We modified the classifier procedure and added protocol validation for detecting exploits that appear in the general protocol and malicious commands that follows normal requests in a connection. Therefore, we used connection tracking to assign state-checking functions to flow packets. Experimental results show that the modified classifier succeeds in detecting a higher number of exploits and malicious traffic than the conventional version does. To detect inter-VM communication in the virtualization platform, we set up our classifier on the virtual network of a Xen system. The bridge-Netfilter was modified to enable the PROMISC hook of the Netfilter for packet processing because supports connection tracking. The classifier connected to the PROMISC hook receives packets and returns them to bridge after pattern matching. We experimentally demonstrated that our classifier performs more accurately than another IDS does. Because only the hypervisor of the Xen platform is modified, the modified classifier can be set up on other virtualization platforms, such as KVM. In future work, we aim to manage several IDSs when more than one is deployed, with experimental testing in a practical scenario being our objective.

Acknowledgement. This research was supported by a grant from the Ministry of Science and Technology, Taiwan, Republic of China, under Grants MOST-103-2221-E-006-145-MY3 and MOST-103-2811-E-006-049, for which we are grateful.

References

1. Netfilter. http://www.netfilter.org
2. Libpcap. http://www.tcpdump.org
3. Snort. http://www.snort.org
4. The Bro Network Security Monitor. http://www.bro.org
5. Wehrle, K., Pählke, F., Ritter, H., Müller, D., Bechler, M.: The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel (2004)
6. Qi, Y., Xu, L., Yang, B., Xue, Y., Li, J.: Packet classification algorithms: from theory to practice. In: INFOCOM 2009, pp. 648–656. IEEE (2009)
7. Finsterbusch, M., Richter, C., Rocha, E., Muller, J.: A survey of payload-based traffic classification approaches. In: IEEE Communications Surverys & Tutorials (2012)
8. Sicker, D.C., Ohm, P., Grunwald, D.: Legal issues surrounding monitoring during network research. In: Proceedings of the 7th ACM SIGCOMM on Internet Measurement (2007)
9. Rotsos, C., Van Gael, J., Moore, A.W., Ghahramani, Z.: Probabilistic graphical models for semi-supervised traffic classification. In: Proceedings of the 6th International Wireless Communications and Mobile Computing Conference, pp. 752–757 (2010)
10. Piskac, P., Novotny, J.: Using of time characteristics in data flow for traffic classification. In: Proceedings of the Autonomous Infrastructure, Management, and Security: Managing the Dynamics of Networks and Services, pp. 173–176 (2011)
11. Lin, P.C., Li, Z.X., Lin, Y.D., Lai, Y.C., Lin, F.: Profiling and accelerating string matching algorithms in three network content security applications. IEEE Commun. Surv. Tutorials **8**(2), 24–37 (2006)
12. Liu, C., Wu, J.: Fast deep packet inspection with a dual finite automata. IEEE Trans. Comput. **62**(2), 310–321 (2013)
13. Wang, X., Jiang, J., Tang, Y., Liu, B., Wang, X.: StriD2FA: scalable regular expression matching for deep packet inspection. In: 2011 IEEE ICC Conference, pp. 1–5 (2011)
14. Risso, F., Baldi, M., Morandi, O., Baldini, A., Monclus, P.: Lightweight, payload-based traffic classification: an experimental evaluation. In: IEEE ICC Conference (2008)
15. Liao, M.Y., Luo, M.Y., Yang, C.S., Chen, C.H., Wu, P.C., Chen, Y.C.: Design and evaluation of deep packet inspection system: a case study. IET Netw. **1**, 2–9 (2012)
16. KVM. http://www.linux-kvm.org/page/Main_Page
17. Xen. http://www.xenproject.org/
18. Khoudali, S., Benzidane, K., Sekkaki, A.: Inter-VM packet inspection in cloud computing. In: Communications, Computers and Applications (MIC-CCA) (2012)

19. Lee, J.H., Park, M.W., Eon, J.H., Chung, T.M.: Multilevel intrusion detection system and log management in cloud computing. In: ICACT (2011)
20. Wu, H., Yi, D., Winer, C., Li, Y.: Network security for virtual machine in cloud computing. In: ICCIT 2010 (2010)
21. Jin, H., Xi, G.F., Zou, D.Q., Wu, S., Zhao, F., Li, M., Zheng, W.: A VMM-based intrusion prevention system in cloud computing environment. J. Supercomputing **66**, 1133–1151 (2013)