Hindawi
Security and Communication Networks
Volume 2020, Article ID 8538519, 16 pages
https://doi.org/10.1155/2020/8538519

Research Article

Extension of Research on Security as a Service for VMs in IaaS Platform

Xueyuan Yin,[1] **Xingshu Chen**,[2] **Lin Chen**,[1] **and Hui Li**[1]

[1]*College of Computer Science, Sichuan University, Chengdu 610065, China*
[2]*Cybersecurity Research Institute, Sichuan University, Chengdu 610065, China*

Correspondence should be addressed to Xingshu Chen; chenxsh@scu.edu.cn

Received 31 October 2019; Revised 7 January 2020; Accepted 21 January 2020; Published 22 April 2020

Guest Editor: Veljko Milutinovic

To satisfy security concerns including infrastructure as a service (IaaS) security framework, security service access, network anomaly detection, and virtual machine (VM) monitoring, a layered security framework is built which composes of a physical layer, a virtualization layer, and a security management layer. Then, two security service access methods are realized for various security tools from the perspective of whether security tools generate communication traffic. One without generating traffic employs the VM traffic redirection technology and the other leveraged the mechanism of multitasking process access. Moreover, a stacked LSTM-based network anomaly detection agentless method is proposed, which has advantages of a higher ratio of precision and recall. Finally, a Hypervisor-based agentless monitoring method for VMs based on dynamic code injection is proposed, which has benefits of high security of the external monitoring method and good context analysis of the internal monitoring mechanism. The experimental results demonstrate the effectiveness of the proposed protection framework and the corresponding security mechanisms, respectively.

1. Introduction

To satisfy partial security requirements including security framework, security service access, network anomaly detection, and monitoring for VMs in IaaS environment [1, 2], various designs have been proposed.

Primarily, a comprehensive security framework for the IaaS platform needs to be considered. CSA proposes a trusted cloud initiative reference architecture [2, 3], which contains BOSS, ITOS, SRM, and a hierarchical cloud service. NIST proposes a role-based cloud architecture [4], which defines capabilities and responsibilities of cloud providers, cloud users, cloud auditors, and cloud agents, respectively. ENISA issues a security framework for the government cloud [5], which abstracts the role elements (including cloud users, cloud owners, and cloud providers) and gives a cloud security framework based on the PDCA (Plan, Do, Check, Act) lifecycle. Amazon (AWS) builds a cloud model based on shared security responsibilities that AWS takes responsibility for the common infrastructure and underlying services, and users undertake responsibility for their own applications and contents. Alibaba, Tencent, and H3C build cloud protection systems in manners similar to AWS. OpenStack and OpenNebula rely on user authentication and communication access control to ensure VM security, which lacks flexible mechanisms to access security services. Varadharajan and Tupakula [6] analyse different threats from tenant administrators, tenants, cloud provider administrators, and the Internet based on Eucalyptus IaaS platform and builds a security service model by adding security components at different component levels. Lin et al. [7] points out that cloud security needed to rely on trusted roots, trusted links, security management, and security evaluation elements. Currently, many research results focus on abstracting cloud security frameworks, which are not conducive to implement cloud security technologies rapidly.

Secondly, VM traffic cannot be monitored or blocked because the virtual communication channel is logical and it is invisibile of traditional security products. To satisfy the requirements, VMware NSX [8] implements a software firewall implanted in the physical computing system kernel in a tightly coupled manner. Sugon [9] proposes a

vulnerability scanning tool that encapsulated in a VM. However, a large amount of storage space is needed as the number of the VM increases, and the characteristic of multitasking parallel processing of the vulnerability scanning engine cannot be brought into play. A security group function implemented in Openstack employs the Netfilter/Iptables to implement a software virtual firewall, which is inconvenient to expand dynamically due to the tight coupling implementation. Cisco and HP develop new protocols of VN-TAG and Virtual Ethernet Port Aggregator (VEPA), respectively, to pull the VM traffic to the physical access layer switch to implement VM communication control [10, 11].

Thirdly, network anomaly detection is a key mean of network security protection, which could analyse network traffic agentless. Currently, network anomaly detection methods can be classified into four categories [12]. (1) Classification method: the method needs security expert which provides a known pattern of an attack and designed detailed features. Hamamoto et al. [13] proposes a method of combining a genetic algorithm and fuzzy logic for network anomaly detection, which uses a genetic algorithm to generate multiple features and decides whether an event is an anomaly by the fuzzy logic method. The results achieve an accuracy of 96.53% and false positive rate of 0.56%. The classification method could make full use of the detailed characteristics of network traffic and obtain high accuracy. (2) Statistics method: the basic idea of the method is to consider a large departure of events from normal as anomalies, thus profiling normal network behavior can detect unknown attacks theoretically. Fernandes et al. [14] propose two anomaly detection mechanisms based on statistical procedure PCA and the ant colony optimization, which profiles normal network behaviours and compares with the real network traffic by a modification of the dynamic time warping metric to recognize anomalous events. However, statistics methods are not suitable for some small-scale attacks due to loss of detailed features. (3) Information theory: information-theoretic measures such as entropy could be used to create an anomaly detection model. Callegari et al. [15] proposes an intrusion detection method that performs anomaly detection by studying the variation in the entropy associated with the network traffic. The traffic is aggregated through three-dimensional reversible sketches and computes the entropy of different traffic descriptors by using several definitions of entropy. Finally, detecting network anomaly according to the deviation of entropy, information-theoretic ignores the detailed information of traffic, which is hard to detect small-scale attacks. (4) Clustering method: the method does not require labelling data and could mine the pattern of data automatically. An evolution of the Microclustering Outlier Detection (MCOD) machine learning algorithm is proposed by Flanagan et al. [16]. The method of distance-based outlier detection and cluster density analysis are combined to detect anomaly in time series, which do not need to label data. However, it is difficult to design distance metrics.

Finally, monitoring is an essential part of the security assurance mechanism [1, 2]. Currently, monitoring technologies for the state of a VM can be classified in two categories. (1) Internal monitoring method (a software-agent is run inside a VM), which provides good context analysis for the state of the VM, but malicious code running inside the VM could damage the monitoring system. (2) External monitoring method (or Hypervisor-based monitoring method) [17], which is realized by employing a Hypervisor or a Virtual Machine Monitor (VMM). The state of a VM can be monitored through some specific Hypervisor APIs, while the retrieved information is limited to the inherent APIs severely without opportunities to expand or customize, such as products of OpenStack Ceilometer and AWS CloudWatch. To make up for that deficiency, the methods based on Virtual Machine Introspection (VMI) have been proposed [18–20]. The methods always rely on setting a specific hardware trapping for internal resource (e.g., CPU or memory) of a VM firstly. When the VM accesses the specific resource, the operation will be captured by the VMM immediately. Then, the introspection mechanism is adopted to monitor the state of the VM. Compared with the internal monitoring method, Hypervisor-based method has a higher safety and reliability, but it consumes more resources. To solve the problem, a triggering method based on VMFUNC is proposed [21], which minimizes overhead of VM exits and avoids the system pause caused by programs of VMI.

To satisfy abovementioned security concerns, a corresponding solution and techniques are designed and realized in the following sections. In Section 2, a cloud security framework compiled the security model of P2DR and the ideas of defense-in-depth of IATF and security as a service was constructed. The framework employed a physical layer, a virtualization layer, and a security management layer. In Section 3.1, to satisfy the requirement of security service access, two access methods were realized for various security tools by leveraging NV, NFV, and SDN technologies from the perspective of whether the tools generate communication traffic. One without generating traffic employed the VM traffic redirection technology with loose coupling and had better flexibility. The other one leveraged the mechanism of multitasking process access and the characteristic of multitasking parallel processing of partial security engine could be brought into play, which was helpful to reduce resource consumption. In Section 3.2, to ensure network security of the VM, a LSTM-based network anomaly detection method was proposed. Based on the periodicity of network traffic, 11 features were selected to describe traffic structure in a time window. Then, multivariate time series consisting of 11 features were input to LSTM-NAD for training. Finally, the finely trained LSTM-NAD model predicted a binary value, which represents whether the current time window was abnormal or not. Compared with paper [1], the LSTM-NAD method had the advantages of higher AUC score and detecting anomaly more effectively. In Section 3.3, based on the characteristics of the internal monitoring method and external monitoring mechanism, a Hypervisor-based agentless monitoring method for VMs based on the mechanism of dynamic code injection was proposed. The proposed mechanism had benefits of high security of the external monitoring method that could not be destroyed by malicious code running inside the VM and had the virtues of good context analysis for the state of the VM, low resource

overhead, and rapid response. Compared to the virtualization layer monitoring mechanism that completely relied on semantic analysis, the method had the advantages of lower performance overhead and faster response. In Section 4, the experimental campaign and discussion about the effectiveness and performance costs of the proposed framework and the various supporting securities were given. More complete technical discussions about the framework and concrete technologies (e.g., Encryption, VPN, VM escape detection, data backup, and Software update) will be discussed in future work.

2. IaaS Protection Framework

To satisfy security requirements of network access control, network anomaly detection, and VM monitoring for the physical and virtual environment of IaaS environment, an extensible security protective framework for the IaaS platform was built, as shown in Figure 1.

The framework complies the security model of P2DR (Policy, Protection, Detection, Response) and ideas of defense-in-depth of information assurance technical framework (IATF) and security as a service. Overall, the framework consists of a physical layer, a virtualization layer, and a security management layer. The physical layer provides physical CPU, memory, storage, network resources for an IaaS platform. Then, building on top of it, the virtualization layer employs Xen, KVM, Hyper-V, LVM, Software Defined Networking (SDN), Network Function Virtualization (NFV), and Network Virtualization (NV) to virtualize physical resources. Moreover, both layers satisfy constraints of the P2DR model.

For the physical layer, high levels of network isolation between different networks within IaaS environment are provided in the horizontal direction. These segment networks are composed of a Demilitarized Zone (DMZ), a management domain, a VM migration domain, a storage domain, a tenant business domain, and a support domain. Thereinto, the DMZ is a secure communication zone between an external network domain and an interior of the IaaS environment. The management domain allows the cloud provider to manage and monitor resources within IaaS environment. The reason for isolating both cloud migration and storage domains are two-fold: performance, as both traffic need very fast data rates, and security, by fully isolating this network. Tenant business domain hosts virtual resources encapsulated in a VM format. The support domain is consisted of two subdomains, one for the physical layer and the other for the virtualization layer. For the one for physical layer, security measures (e.g., IPS, IDS, traditional physical firewalls, and antivirus gateways) can be deployed in appropriate locations to protect the physical environment.

The virtualization layer contains a tenant business domain and a security support subdomain for the virtualization environment. (1) For the tenant business domain, from the outside to the inside of the tenant VM, a three-layered isolation network perimeter is organized: (L1) Tenant domain (TD) perimeter. (L2) Tenant subdomain perimeter. (L3) VM perimeter. (2) For the security support subdomain, some measures are orchestrated to eliminate partial security risks including resource access control, communication control, network abnormality detection, and VM monitoring. For L1 and L2, capacities of managing tenant VMs communication access control, network abnormal detection, and deep packet inspection can be provided. For L3, services of VM monitoring and file antivirus for VMs can be provided [1].

Based on the abovementioned discussion and combined mechanisms of user identity management, anomaly detection, vulnerability detection, monitoring, data backup, emergency response, etc., a dynamic security protection framework can be built, as shown in Figure 2.

3. Security Service Implementations

3.1. Security Service Access. Currently, most of security tools such as the firewall and vulnerability scanning can be roughly divided into two types from the perspective of whether security service tools generate communication traffic: (1) Type 1: the tool does not generate traffic and only processes and forwards the received traffic (e.g., the firewall filters the traffic passed through). (2) Type 2: the tool generates a communication flow and determines the security status based on the reply information (e.g., vulnerability scanning and port scanning tool). To solve the problem of the security tool is difficult to access the tenant virtual network dynamically, the following concerns should be considered: (1) the implementation form of security service tools. (2) The way to send a VM traffic to a security tool, or the way to deliver a communication flow from a security service tool to a tenant network.

For concern (1), the NFV technology is employed to realize security appliances (e.g., choose a lightweight VM as a carrier for a service of network anomaly detection). Moreover, by integrating multiple security service tools in one security appliance, the security service property of "Encapsulate multiples into one" can be realized.

For concern (2),

(a) When a VM traffic needs to be sent to a security appliance (definition in Type 1), the security appliance should be instantiated in the tenant subnet, thus the network of Layer 2 could be reachable between them. After that, a software switch supporting the OpenFlow protocol is deployed at the VM network interface, which could redirect a packet from a specific VM to a specific security appliance by modifying the native destination MAC address in the packet to the MAC address of the security appliance according the security policy. Then, the specific traffic can be processed or filtered, as shown in Figure 3. Besides, compared with the way of instantiating a security appliance directly into a compute node in a tenant business domain, by instantiating it into a separate support domain, which has no effect on computing resources for VMs, and is more conducive to improve the security service capability.

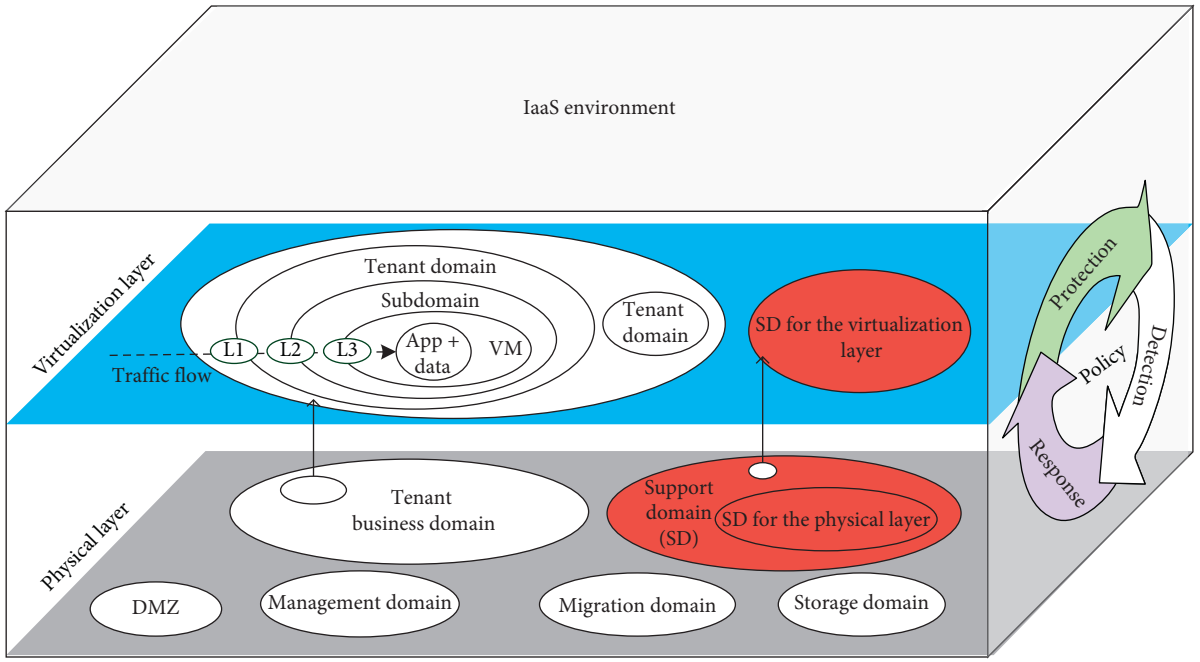

Figure 1: The layered IaaS protection framework model.

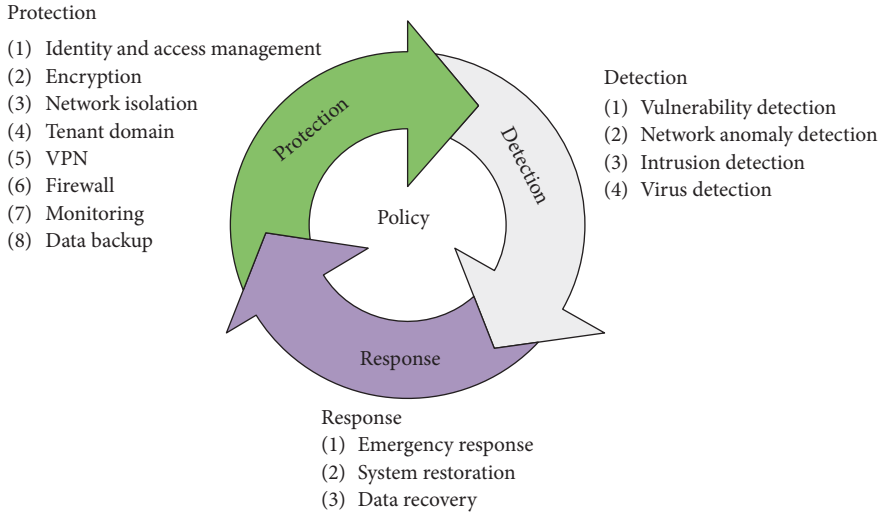

Figure 2: Security protections for IaaS platform complied with the P2DR model.

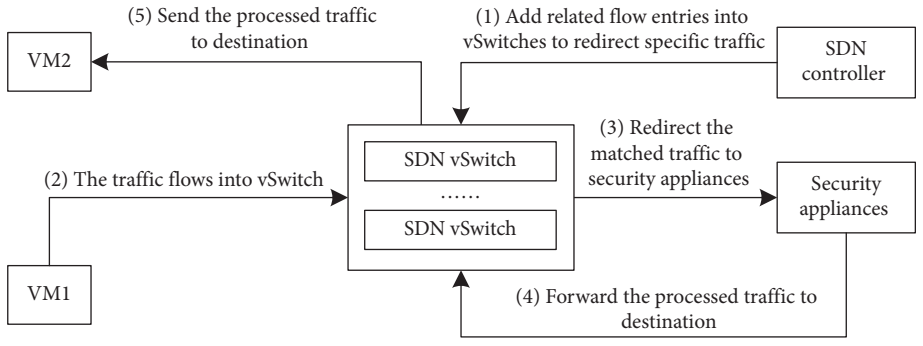

Figure 3: VM traffic redirection mechanism.

(b) When a communication flow from a security appliance (definition in Type 2) is needed to be delivered to a tenant network actively, existing partial approaches based on the VM encapsulation underutilizes the capacity of multitasking parallel processing of the security engine and result in excessive resource consumption.

To solve the problem, a security service access method based on multitask process and SDN is proposed, as shown in Figure 4. By providing an independent security service node and constructing virtual network links dynamically with SDN technology, a single security appliance could serve multiple tenants simultaneously, and the security service property of "one serves more than one" can be realized.

Take a security appliance as a vulnerability scanning as an example to explain the principles of the type 2 security service access method.

(1) According to a security policy, an operation of vulnerability scanning for VM-A is launched. After receiving the specific instruction, the security service creates a task running space for the security service engine and the security task process.

(2) A communication link is established between the task running space and the virtual network device by creating virtual ports and virtual links. Furthermore, the routing is set according to the IP address information of the VM to be scanned, and the communication isolation is implemented by combining the tenant layer network tag of the tenant domain. (a) Virtual ports and links are implemented based on VETH devices. One end of the virtual link is bound to the virtual switch, and the other end is bound to the task running space through the task interface. At the same time, the security task running space is reachable to the target VM network by configuring the IP address of the task interface and the routing information. (b) Communication isolation between different security task processes is implemented based on SDN technology. The network space of the task process is connected to the virtual switch, the Layer 2 network identifier of the tenant domain, such as VLAN TAG, is combined to tag the security service traffic and delivered into the target VM network. At the same time, it guarantees the Layer 2 network isolation between different security task processes. (c) In a virtual network environment, VMs in different tenant domains may be configured with the same IP address. To ensure that different security task traffic could be delivered to a correct target network, virtual IP addresses are set for the security task process and the target VM. The detailed process is illustrated as follows.

The key parameters are the real MAC and IP information of VMA is (*vm.mac, vm.ip*), and the virtual address information mapped to the task running space is (*vir.mac, vir.ip*). The virtual object builds on the VETH device and bounds to the task running space is named *vport*, whose address information is (*vport.mac, vport.ip*). The gateway MAC address of the virtual network where VMA located is *gw.mac*.

(1) The security task process sends packets:

S1: the security task process encapsulates the security task traffic according to the address information configured by the local virtual network interface *vport* and the virtual address information of the target VM. Select a reserved address in the network address as the virtual address resource pool for *vport.ip* and *vir.ip*. At this point, the basic information of the packet is

(SRC = (vport.mac,vport.ip),DST = (vir.mac,vir.ip), TAG = null)

S2: after the data packet reaches at the virtual switch, the security task process data packet is tagged according to the VMA VLAN TAG, and the destination address is modified back to the real target VM address. At this point, the basic information of the packet is

(SRC = (vport.mac,vport.ip), DST = (vm.mac,vm.ip),TAG = ID)

S3: after receiving the security service traffic with the destination address of the VMA and the VLAN tagged equal to VMA VLAN TAG, the VMA access layer virtual switch removes the tag and sends it to the VMA. At this point, the basic information of the packet is

(SRC = (vport.mac,vport.ip), DST = (vm.mac,vm.ip),TAG = null)

(2) The security task process receives packets:

R1: the VMA encapsulates the reply information according to the local address and the saved security task process address. Since the task process and the VM are in different networks, the gateway address is used to reply to the packet. The basic information of the reply packet is

(SRC = (vm.mac,vm.ip),DST = (gw.mac,vport.ip), TAG = null)

R2: after the packet arrives at the virtual switch, tag the traffic that the destination IP address matched to the security task process and modify the destination MAC address to redirect to the security task process interface *vport*. At this point, the basic information of the packet is

(SRC = (vm.mac,vm.ip), DST = (vport.mac,vport.ip),TAG = ID)

R3: after the packet arrives at the virtual switch that the security task process is connected, modify the source IP and source MAC of the traffic flow that destination IP address is equal to the address of the security task process back to the virtual IP address and virtual MAC address. At this point, the basic information of the packet is

(SRC = (vir.mac,vir.ip), DST = (vport.mac,vport.ip),TAG = null)

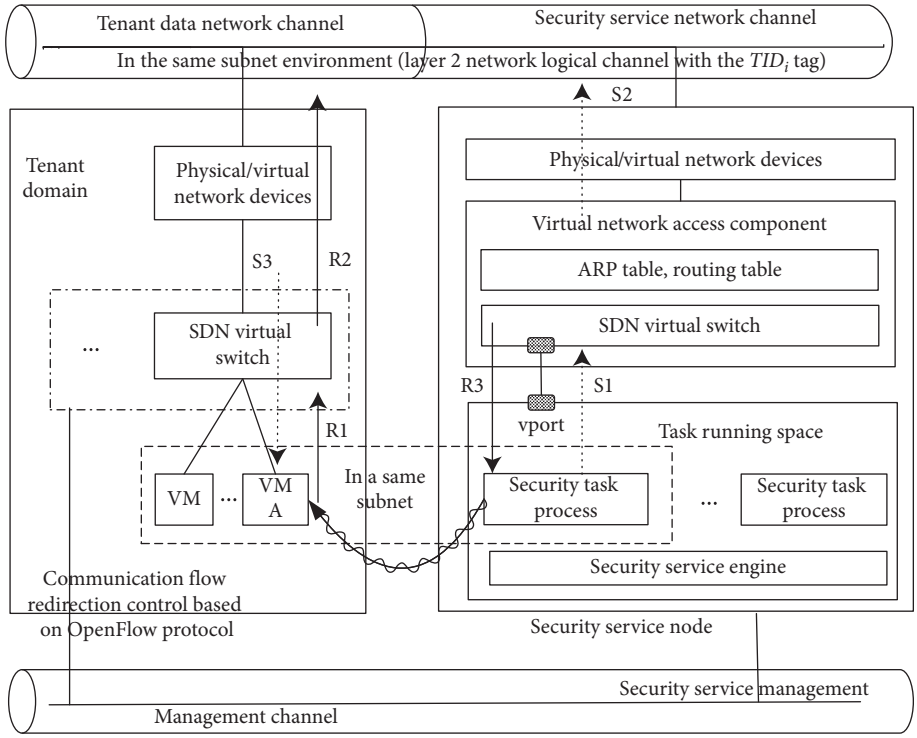

Figure 4: Type 2 security service access model.

By employing the method, the characteristic of multitasking parallel processing of partial security engine can be brought into play, which is helpful to reduce resource consumption that the storage required by the vulnerability scan engine and plugins, which does not increase as the number of security service increased.

3.2. Network Anomaly Detection

3.2.1. Network Traffic Features. Network traffic features are crucial to the mechanism of network anomaly detection, which describe both the normal and abnormal network traffic patterns. Normal network activity of the VM host in the network causes periodic fluctuation of network traffic, thus the static network feature is hard to profile the normal network behavior in a network environment. Many features are time varying and show strong periodic, which are called as the network traffic dynamic structure features.

As shown in Table 1, the 1st to 4th features are the same as paper [1], which are selected to describe the traffic structure in a particular window. Besides, the features 5th to 11th are proposed to describe the dynamic network traffic profile of a VM. Therefore, instead of using the combined features in the time window, original features are adopted, which completely contain all the original information and are beneficial to the advanced characteristics of automatic learning of the recurrent neural network.

The number of VMs in cloud environment is very large, which means that the network traffic is huge, and requires that the features we designed must be easy to collect, the amount of calculation is small, and the payload is invisible due to encrypted traffic and user privacy restrictions. As an example, the network traffic of a web server in a cloud data center is captured, and 11 features in each time windows (which set as 1 minute) are calculated. The statistical results of the abovementioned 11 features in 7 days (total of 10080 time windows) are shown in Figure 5, most of the features show strong periodicity, such as *bps* and *pps*. It is worth noting that some features do not exhibit synchronization, such as *syn_rate* and *bytes_per_ip*, which are almost the opposite trend. Additionally, some outliers are obvious, such as an obvious anomaly in *syn_count*. However, due to the powerful predictive ability of the recurrent neural network, these obvious outliers do not need to be processed, and the network will learn those abnormal patterns.

3.2.2. LSTM-NAD: LSTM-Based Network Anomaly Detection. F is defined as a set of network behavior consisting of 11 features:

$F = \{access\_port,\ ip\_entro,\ bps,\ pps,\ packet\_per\_ip,\ ip\_count,\ service\_port\_entro,\ syn\_rate,\ syn\_per\_ip,\ bytes\_per\_ip,\ syn\_count\}$.

Given a $f_i \in F$, multivariate time series are assumed as $X = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$, and $x^{(t)}$ was an m-dimensional vector $\{x_1^{(t)}, x_2^{(t)}, \ldots, x_m^{(t)}\}$, where each point $x_i^{(t)}$ denotes value of f_i between t windows.

Long short-term memory (LSTM) network can capture the temporal structure of the sequence, which is very suitable for network traffic time series of the VM. The LSTM is a variant of RNN that is capable of learning long term

Table 1: Description of 11 traffic structure features.

ID	Feature name	Description
1	*access_port*	The entropy of port count accessed by each client IP and corresponding probability of occurrence
2	*ip_entro*	Entropy with proportion of every communicated IP
3	*service_port_entro*	The entropy of access service port
4	*syn_rate*	The proportion of packets with SYN flag in all packets
5	*bps*	Bytes per seconds of VM
6	*pps*	Packets per seconds of VM
7	*packet_per_ip*	Average packet counts of every IP
8	*ip_count*	Communicate IP count between every time window
9	*syn_per_ip*	The count of packets with SYN flag per IP
10	*bytes_per_ip*	Average bytes per IP
11	*syn_count*	Count of packet with SYN flag

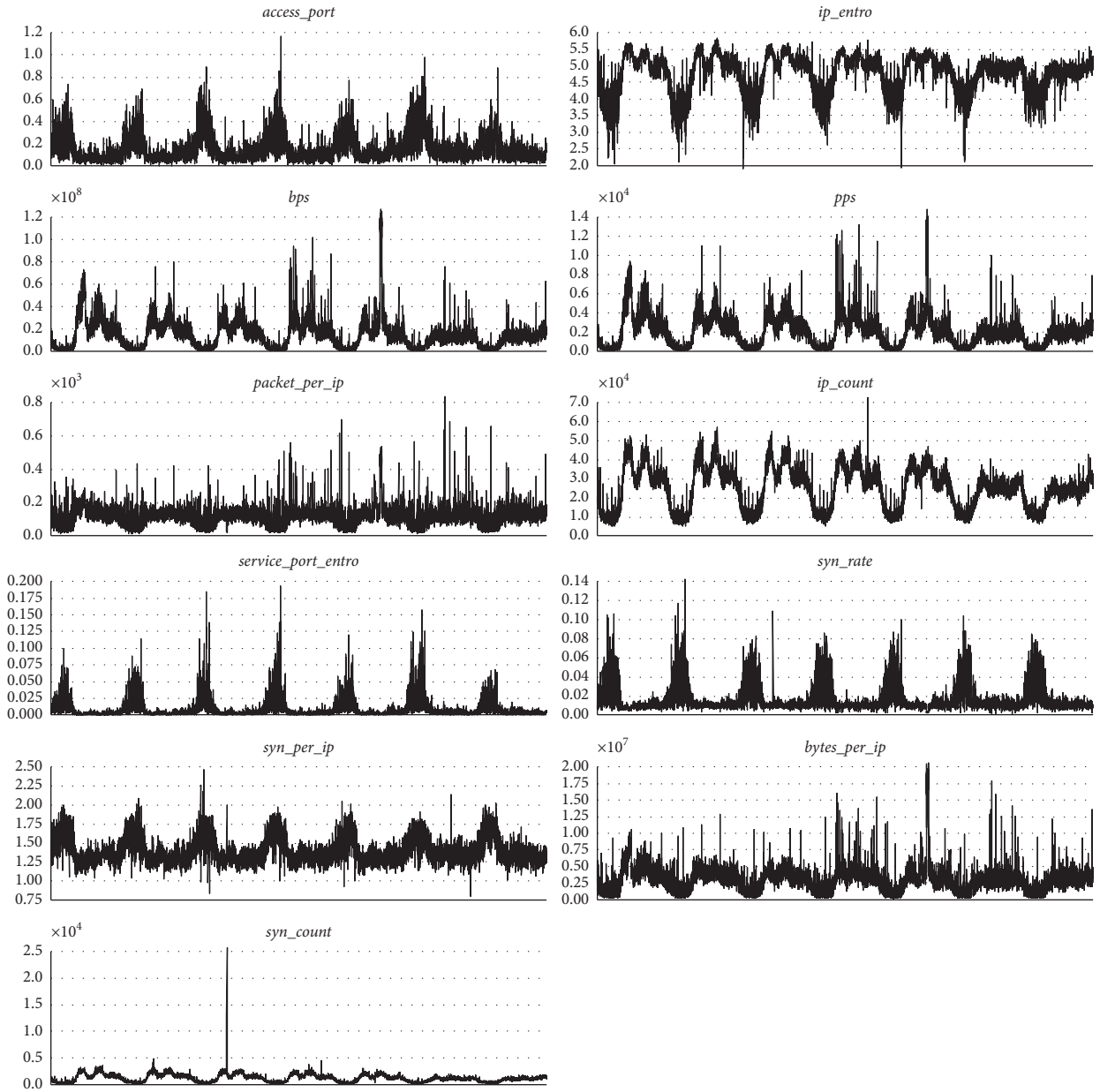

Figure 5: Total 11 features of network traffic.

dependencies, which solves the vanishing gradients problem effectively and is the most widely used type of RNN. The formulas for a single LSTM block are given below [22]:

$$\begin{aligned} i_t &= f_i\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right), \\ z_t &= f_z\left(W_{xz}x_t + W_{hz}h_{t-1} + b_z\right), \\ c_t &= i_t \odot k_t + z_t \odot c_{t-1}, \\ o_t &= f_o\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right), \\ h_t &= o_t \odot f_h\left(c_t\right), \end{aligned} \tag{1}$$

where k_t is an input; i_t is an input gate, which defines a newly computed state for k_t to be permitted pass through; z_t is a forget gate, which defines the previous state that would be permitted pass through; c_t is a cell state, which combines the previous memory c_{t-1} and the new input $i_t \odot k_t$; o_t denotes an output gate; h_t is a hidden state; $\odot$ is an elementwise multiplication; and f is an activation function, which is *sigmoid* or *tanh* function usually.

Therefore, the LSTM network is introduced as a strong classifier to decide whether the current time window is abnormal in the network traffic detection system. Compared with traditional machine learning algorithms, LSTM does not need to extract features elaborately, which achieves end-to-end anomaly detection.

In order to improve the ability of LSTM, a stacked LSTM network framework called "LSTM-NAD" was designed, and the framework is shown in Figure 6. The input shape of the network is an array which shapes like <B, T, F>, B represents the batch size, T represents the time step, and F represents the feature size. Then, 2 stacked LSTM are applied to the input layer, and LSTM units in a hidden layer are fully connected through recurrent connections. The first LSTM layer had 64 hidden units, which outputs the result of each time step, and the second has 32 hidden units, which only outputs value of the last time step. Finally, 2 fully connected layers are followed by LSTM layers, the first layer has 64 units and the second has 32 units, and the last layer in the network is a fully connected layer in which the hidden unit size is 1 and activation function is the *sigmoid* function.

In multivariate time series, the current time window is dependent on the previous windows, so the sliding window is used to generate training sequence data. To improve the training effect, we also shuffle the training data. Since the training samples are extremely unbalanced, the number of abnormal points is very small compared to the number of normal samples. Some data enhancement and oversampling methods are adopted to make the number of positive and negative samples in the training samples roughly equal. Finally, some dropout layers are added to the fully connected layer in order to avoid overfitting.

3.3. VM Monitoring. A Hypervisor-based agentless monitoring method for VMs based on the mechanism of dynamic code injection is proposed, as shown in Figure 7.

Take KVM as a VMM to illustrate. The framework is constructed according to the following reasons:

(1) There is a semantic gap between the KVM and the VM. The monitoring code injected into the VM through KVM runs correctly, and the following requirements must be satisfied: (a) obtain the semantics (the kernel symbol table) of the VM and resolve the kernel symbols used by the monitoring code. (b) Allocate a kernel memory chunk of the VM (named mem-A) to store the monitoring code. (c) After loading the monitoring code into the memory in (b), an operation of relocation repairation is required to perform to make the monitoring code run correctly.

(2) It is necessary to construct an execution opportunity for the injected monitoring code (e.g., by means of intercepting system calls of the VM transparently to create an executive opportunity, the monitoring mechanism will be activated when a system call event occurrence in the VM).

(3) A protection mechanism for the monitoring code also should be considered. Because the monitoring code is hosted in the kernel space of the VM, other kernel modules of the VM may bypass the monitoring mechanism (e.g., by means of (a) hook the SYSENTER_EIP_MSR (MSR) register of the VM that saves the address of the system call entry. (b) Hook the system call table of the VM.

The framework contains (1) monitoring code injection, which is deployed in the KVM to perform an injection action for the monitoring code, when a VM (named VM-A) is powered on and modifying the monitored system call entry to the address of corresponding function of the monitoring code transparently. After that, the system call execution process of the VM-A could be monitored. (2) Monitoring protection, which is deployed in the KVM to accomplish read-only protection for the injected code, the MSR register, the system call entry function, and the system call table. After that, they could not be tampered by the malicious code running inside of the VM-A. (3) A shared memory (named mem-S), which is allocated at the first run of the monitoring code, and its base address and size will be passed to the KVM by means of VMCALL of the super system call. Thus, the monitoring code will interact with the KVM through the mem-A to exchange data (including delivering policies and retrieving results). (4) Monitoring policies, which include IDs of the monitored VMs, system call numbers to be monitored, and the system call response actions (e.g., record or block an operation of placing sensitive information in a file inside a VM).

3.3.1. Monitoring Code Injection. KVM injects the monitoring code into VM-A dynamically by the following steps: Step (1) When VM-A is powered on, the kernel symbol table of VM-A is resolved. Step (2) A kernel memory area (named mem-A) of VM-A is allocated. Step (3) According to the kernel symbol table of VM-A and the base address of mem-A, the symbol table of the monitoring code is resolved and the relocation data is repaired. Step (4) The reconstructed monitoring code is injected into mem-A finally.

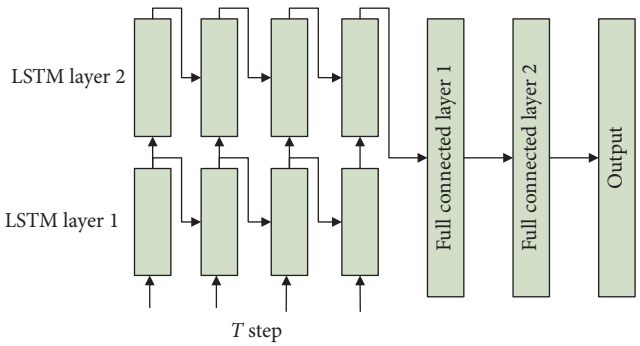

Figure 6: The framework of LSTM-NAD.

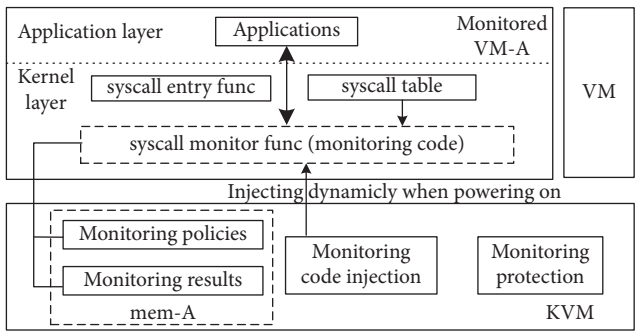

Figure 7: The framework of the agentless monitoring system.

(1) VM kernel Symbol Resolution. Take CentOS 7.2 x86_64 OS as an example to illustrate the mechanism. Figure 8 shows the distribution of the kernel symbol table of CentOS.

From Figure 8, *kallsyms_addresses* is an array that stores the symbol name and the corresponding kernel virtual address, which is hosted in the read-only data segment and arranged in ascending order; *kallsyms_num_syms* stores the number of kernel symbols, which is equal to the entry number of *kallsyms_addresses* and also the first descending value; *kallsyms_names* is an array that stores all kernel symbol strings after being ASCII encoded, which includes *kallsyms_num_syms* entries in total; *kallsyms_markers* is an array that stores the kernel symbol offset after *kallsyms_names* is grouped (every adjacent 256 kernel symbols are divided into a group); *kallsyms_token_table* is an array that stores the strings which are commonly used; *kallsyms_token_index* is an array that stores the offset of every entry of *kallsyms_names* in *kallsyms_token_table*.

According to the distribution rule of Figure 8, the memory of code segment of VM-A was searched from the low address to the high address in the KVM to analyze the addresses of the key symbols and the kernel symbol table (named VM_SYMS) of VM-A.

(2) VM Kernel Memory Allocation. KVM needs to call the function of *module_alloc* (a kernel memory allocation function) of VM-A actively to allocate a mem-A. Based on the kernel symbol resolution method, after the KVM gets the address of *module_alloc* of VM-A, we employ a method of bottom-up function call channel to allocate a mem-A from the high address of the kernel memory area of VM-A. The detailed

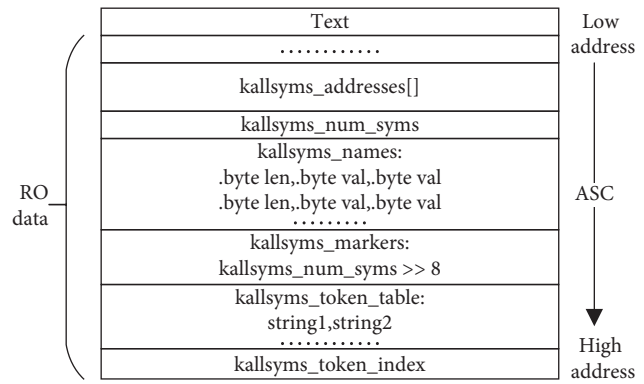

Figure 8: Distribution of the system kernel symbol table of CentOS.

steps are as follows: Step (1) Save the information of current execution environment of VM-A (e.g., values of instruction registers) when an event occurrence with an exit of the VM-A. Step (2) Construct parameters for the *module_alloc* through the instruction register. Step (3) Modify the value of the instruction register to the address of *module_alloc*. Step (4) The exit occurrence in Step (1) would reoccur after executing the *module_alloc* and restore the information saved in Step (1).

(3) Monitoring Code Symbol Resolution and Relocation Data Reparation. The principle of monitoring code symbol resolution is shown in Algorithm 1. Every entry in the symbol table of the monitoring code will be traversed, if the address of *sym[i]* is NULL, the address will be retrieved and reassigned by looking up VM_SYMS of VM-A. Else, it will be amended according to the base address of kernel memory of VM-A and its relative offset.

In Algorithm 1, *symsec* is a pointer to a symbol table section in the ELF format file of the monitoring code; *sym* is the symbol table of the ELF format file. Every entry in the table contains information of an index (*st_name*), a type (*st_shndx*), and an address (*st_value*) of every kernel symbol name in *strtab*; *strtab* is a pointer to a string table; SHN_UNDEF is an unresolved symbol; *guest_load_addr* is a base address of VM-A kernel memory that the KVM has allocated; *sechdrs* is a pointer to the section head table of the ELF format file. Every entry in the table stored the information including offset (*sh_offset*) relative to *sechdrs*, section size (*sh_size*), and code segment address (*sh_info*); *name* is a temporary variable that stores a name of a kernel symbol; *get_address (name)* is the function of according to name variable to retrieve the address from VM_SYMS.

The principle of relocation data repairation is shown in Algorithm 2, which contains the following steps: Step (1) Traverse the relocation table to retrieve every entry information (including an index of the current entry and an addressing type (e.g., R_X86_64_PC32) of the entry to be relocated). Step (2) Calculate the address of the entry to be relocated in memory of VM-A according to the relocation type and the resolved symbol table and then write it into the relocation table. Step (3) Load the address of the reconstructed relocation table to the address of *guest_load_addr*.

```
Input: symsec, sym, strtab, guest_load_addr
Output: Symbol resolved code in memory
for i ⟵ 1 to symsec->sh_size/sizeof(Elf_Sym) do
    name ⟵ strtab + sym[i].st_name;
    if sym[i].st_shndx = SHN_UNDEF then
        sym[i].st_value ⟵ get_ address(name);
    else
        sym[i].st_value ⟵ sym[i].st_value +
            guest_load_addr +
            sechdrs[sym[i].st_shndx].sh_offset;
        end
end
```

Algorithm 1: GuestCodeSymResv.

```
Input: hdr, sechdrs, relsec, symindex,strtab, guest_load_addr
Output: Relocated resolved code in memory
rel ⟵ hdr + sechdrs[relsec].sh_offset;
for i ⟵ 0 to sechdrs[relsec].sh_size/sizeof(*rel) do
    target_sec ⟵ sechdrs[relsec].sh_info;
    loc ⟵ hdr + sechdrs[target_sec].sh_offset + rel[i].r_offset;
        fake_loc⟵guest_load_addr +
            sech drs[target_sec].sh_offset + rel[i].r_offset;
        sym ⟵ hdr + sechdrs[symindex].sh_offset +
            ELF_R_SYM(rel[i].r_info);
    symname ⟵ strtab + sym->st_name;
    val ⟵ sym->st_value + rel[i].r_addend;
    rel_type ⟵ ELF_R_TYPE(rel[i].r_info);
    switch rel_type do
        case rel_type = R_X86_64_64*(u64*)loc ⟵ val;
        case rel_type = R_X86_64_32*(u32*)loc ⟵ val;
        case rel_type = R_X86_64_32S*(s32*)loc ⟵ val;
        case rel_type = R_X86_64_PC32 val ⟵ val-(u64)fake_loc; *(u32*)loc ⟵ val;
            otherwise
                Output Unknown;
        endsw
    endsw
end
```

Algorithm 2: GuestCodeRelocResv.

In Algorithm 2, *hdr* is the base address of the monitoring code in ELF format file loaded into a KVM temporary memory; *rel* is, a temporary variable, a pointer to the relocation table section, and every entry in the rel table stores a relative offset (*r_offset*) and offset type information (*r_info*) of the entry (that to be relocated); *relsec* is an index of the relocation section in *sechdrs*; *target_sec* is a temporary variable that points to the code segment of the relocation section; *loc* is a temporary variable that points to the address of *rel[i]* in the KVM temporary memory; *fake_loc* is a temporary variable that points to the address of *rel[i]* in the memory of the VM-A; *symindex* represents an index of the symbol table section in the section header table of the ELF format file; *rel_type* is a temporary variable that stores a value of the relocation type; *val* is a temporary variable that stores a target address after *rel[i]* had been relocated.

(4) Dynamic Injection Monitoring Code. The operation of the injection monitoring code whose symbols have been resolved and repaired can be accomplished by the KVM by writing the monitoring code to *mem-A*. Then, redirect corresponding guest system call execution paths to our monitoring code transparently to make the monitoring mechanism take effect. The steps are as follows: Step (1) According to VM_SYMS, retrieve the address of a monitored system call in the system call table of VM-A. Step (2) Reassign the address value in Step (1) to the value of corresponding monitoring function (*func_name*) in the monitoring code.

3.3.2. Monitoring Function Protections. The protection mechanisms include an antitampering function for the address register (MSR) of system call entry and a read-only

mechanism for the kernel code (including the system call entry function and the injected monitoring code) and the system call table.

(1) Protection for MSR: turn on the write operation trapping of SYSENTER_EIP_MSR of VM-A by setting MSR bitmaps of the KVM. When VM-A attempts to modify the value of SYSENTER_-EIP_MSR, it will be captured by the KVM. Because the KVM has higher privileges, the KVM can invalidate the abovementioned modification operation directly. Moreover, the writing operation to SYSENTER_EIP_MSR only occur when VM-A kernel is initialized, which bring little impact on performance of VM-A.

(2) Read-only protection for the kernel code and the system call table. (A) When VM-A accesses data, the page table management mechanism of the OS converts the virtual address to the physical address of VM-A. Then, the CPU traverses the *ept* page table and converts the physical address of VM-A to the physical address of the host, and the data in memory could be retrieved finally. (B) The page table of VM-A is maintained by itself, and the malicious module inside VM-A can tamper with the kernel. (C) The *ept* page table is maintained by the KVM, which cannot be tampered by the malicious code of VM-A. The protection module achieves security protection by mapping the kernel codes and the system call table of VM-A into a read-only mode in the *ept*.

3.3.3. Monitoring Policy Definition. Monitoring policy P is defined formally as a four-tuple: $P = (\mathrm{ID}, \mathrm{Nr}, \mathrm{Conf}, \mathrm{Action})$.

(1) ID represents the ID of a VM to be monitored, $\mathrm{ID} = \{id_1, \ldots, id_n\}$, if $i \neq k$, then $id_i \neq id_k$.

(2) Nr represents a system call number set to be monitored, $\mathrm{Nr} = \{nr_1, \ldots, nr_m\}$, if $i \neq k$, then $nr_i \neq nr_k$.

(3) Conf represents a system call configuration set (e.g., a blacklist set of the monitored processes), $\mathrm{Conf} = \{(s_1, o_1), \ldots, (s_u, o_v)\}$, the subject S initiates the operation and the object O is being accessed, $S = \{s_1, \ldots, s_w\}, O = \{o_1, \ldots, o_x\}$, if $i \neq k$, then $s_i \neq s_k, o_i \neq o_k$.

(4) Action represents real time responses (e.g., record and block the operation of placing sensitive information in a file on a VM) of the monitored system calls, $\mathrm{Action} = \{\mathrm{record}, \mathrm{block}\}$

4. Experimental Campaign

According to Figure 1, we employ several normal servers, Gigabit Ethernet switches and Gigabit NICs to build the IaaS platform. The key parameters are OpenStack Kilo as the IaaS management toolkit, KVM as the VMM and version is 2.3.0, OpenvSwitch as the virtual SDN switch and Ryu as the SDN controller, and CentOS 7.2 x64 release version as the host OS and its kernel version is 3.10.0, and OpenVAS 7 as the vulnerability scanning engine. The key physical host parameters are the CPU model is Intel (R) Xeon (R) CPU E5-2630 v3 $*$ 2, 2.40 GHz, memory capacity is 128 GB. Other experimental parameters will be given in the corresponding section. The deployment architecture is as shown in Figure 9.

The Tenant Domain is constructed by employing NV, SDN, and NFV technologies. (1) Logical communication channel is constructed by employing the VxLAN technology; (2) subnet configuration is relied on the virtual appliances of DHCP, router, and switch implemented by NFV technology; (3) tenant domain perimeter is built by a set of virtual routers and firewalls those deployed on the logical communication channel; (4) communication access control that leveraging the ACRs configured in virtual firewalls. The isolation and interconnection between different tenant domains and subdomains are realized by employing different virtual routers which configured different subnets and firewalls. The firewall could load iptables/ebtables rules, or SDN forward rules to implement access control. Moreover, the support domain is constructed by leveraging virtual firewall, network anomaly detection, VM monitoring, and VM antivirus [1].

4.1. Security Service Access. For the Type 1 access method, the VM communication delay is increased due to the VM traffic is redirected to the security appliance firstly. The statistics of the PING operation delay test before and after configuring the redirect policy for two VMs in the same subnet in the IaaS environment is shown in Figure 10. The results show that the communication delay after redirection is twice that of in redirect manner, which is as expected.

For the Type 2 access method, which compares with the current method of providing security scanning with a VM granularity, we perform tests of the scanning performance by employing OpenVAS 7 tool and the resource consumed by the multitasks method and the multi-VMs method in scanning different numbers of tenant network domains, as shown in Table 2.

The result shows that providing a vulnerability detection service at the VM granularity for different tenants leads to a sharp increase in resource consumption. Moreover, because the method proposes in this paper involved virtual network device creation, network address translation, etc., statistically, which only increases 0.1 ms delay compared with the way the VM provides services directly.

4.2. Network Anomaly Detection. To prove the effectiveness of the proposed method, an experiment is designed to compare the detection results of our algorithm and paper [1] method in a real network environment that is similar to paper [1]. The network traffic of a web server running in a VM is captured and 11 features of network traffic are monitored for a month, which contains about 37439 time windows, and the first 21600 records are for training and the rest for testing.

The network traffic dataset contains about 2.8 billion packets in train dataset and about 1.6 billion packets in test dataset. In order to distinguish normal events and abnormal events, we analyze network traffic and system logs about the server and

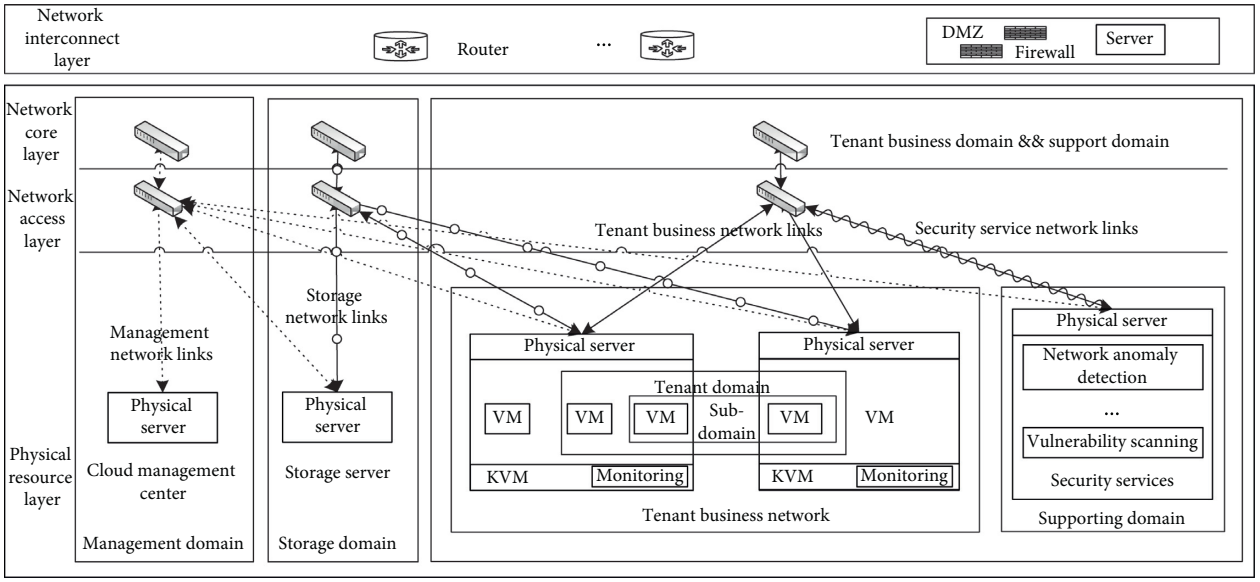

Figure 9: The deployment architecture of the IaaS platform.

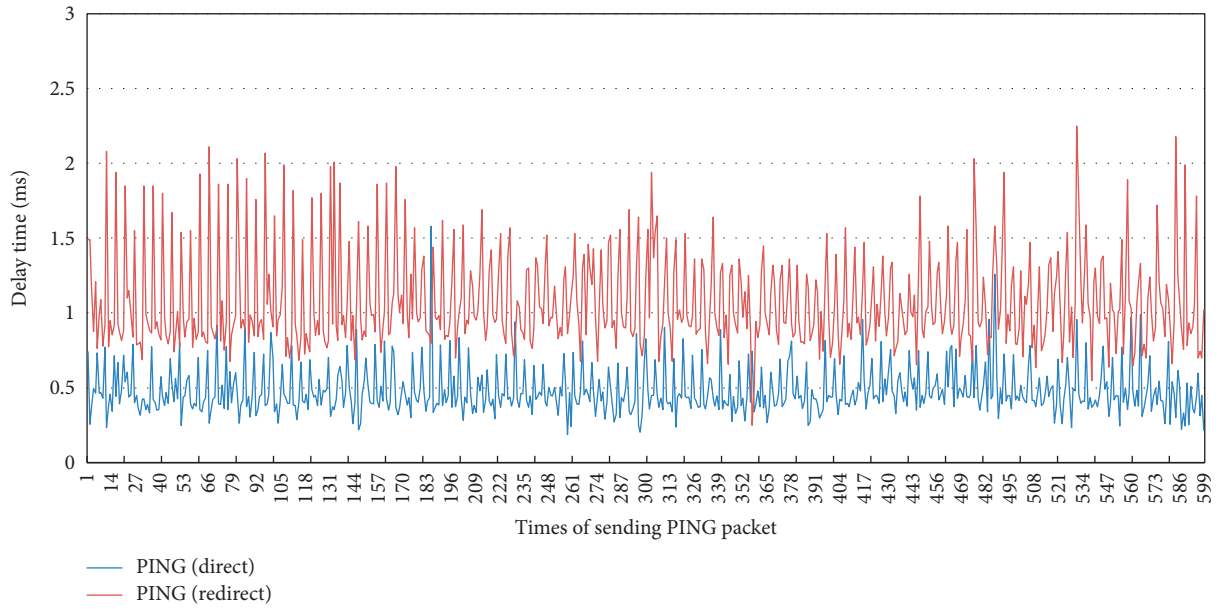

Figure 10: Consumed time of PING operation without/with VM communication redirection.

Table 2: Performance test result (%).

Test items	Number of tenants	Multi-VMs method	Multitasking method
CPU	1	22.3	14.6
	3	64.2	22.7
	5	99.8	35.5
Memory	1	14.1	2.3
	3	35.7	6.1
	5	73.5	13.9

mark abnormal and normal manually, which contain SYN flood, URL probe, UDP flood, port scan, and partial unknown attacks. Details of events are as shown in Table 3.

After numerous experiments, the LSTM-NAD network usually converges in 50 epochs, so the epoch is set to 50 and the callback function is set to save the best model. Since

Table 3: Details of events.

Datasets	Total events	Normal event	Abnormal event
Train dataset	21600	21497	103
Test dataset	15839	15721	118

network anomaly detection is a binary classification problem, the loss function is set to binary cross entropy. Finally, the LSTM-NAD is implemented by employing Keras and Tensorflow and running on a server configured a NVIDIA Tesla P40 graphics card.

Receiver-operating characteristic (ROC) and the area under curve (AUC) are two of the most important metrics for evaluating any performance of classification. ROC plots describe the False Positive Rate (FPR) versus the True Positive Rate (TPR) over all possible threshold values. If the observed curve in a ROC plot falls along the diagonal line x equals to y, the model is assessed to perform no better than random chance. A perfect curve is the one that forms a right angle at FPR = 0.0 and TPR = 1.0 which will indicate that all assigned probabilities to the positive class (here malicious activity) are greater than assigned probabilities to the negative class (here benign activity). AUC is the value attained by integrating over the ROC curve. An AUC of 1.0 denotes perfect classification, and an AUC of 0.5 denotes random chance.

Thus, those two metrics are used to evaluate the anomaly detection performance of each method, and the ROC-AUC figure for two methods is as shown in Figure 11.

As seen from Figure 11, AUC of LSTM-NAD is 0.94, which is a pretty high score compared with [1] and suggests that our LSTM-NAD methods have an effective performance on network anomaly detection. On the other hand, ROC of LSTM-NAD almost covered [1], which means LSTM-NAD has a higher true positive rate and lower false positive rate regardless of the threshold. In the experiment, [1] detects some unknown anomalies, and the anomalies are also detected by LSTM-NAD, which indicates that LSTM-NAD can detect unknown anomalies thanks to the memory of the normal network traffic pattern.

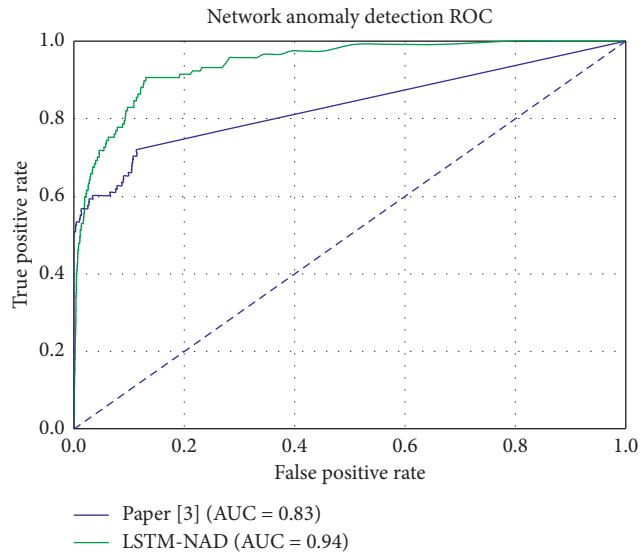

Figure 11: ROC and AUC for two methods.

```
[276230.460696] FILE MONITOR: INDEX:[0], timestamp: [1537193189],filename:[test2],
pathname: [/home/test2], operation: [OPEN], process name: [bash], user id:[0].
[276230.460698] FILE MONITOR: INDEX:[1], timestamp: [1537193189],filename:[test2],
pathname: [/home/test2], operation: [CLOSE], process name: [bash], user id:[0].
[276230.460701] FILE MONITOR: INDEX:[2], timestamp: [1537193190],filename:[test.c],
pathname: [/home/test2/test.c], operation: [OPEN], process name: [cat], user id: [0].
[276230.460703] FILE MONITOR: INDEX:[3], timestamp: [1537193190],filename:[test.c],
pathname: [/home/test2/test.c], operation: [READ], process name: [cat], user id: [0].
[276230.460705] FILE MONITOR: INDEX:[4], timestamp: [1537193190],filename:[test.c],
pathname: [/home/test2/test.c], operation: [CLOSE], process name: [cat], user id: [0].
```

Figure 12: The file operation monitoring result of the test VM.

4.3. VM Monitoring

(1) Functional Validity Verifications. In a VM, the monitored file operation information is viewed from the log of the KVM when opening or editing the monitored files, including the file name, file path, operation process, user ID (uid), and operation type, as shown in Figure 12. Furthermore, because the monitoring code exists only in the kernel memory of the VM and does not depend on the OS driver and the process-related kernel data structures, any information about the monitoring system cannot be detected by using detection tools such as "*lsmod*" or "*ps*" running inside the VM.

(2) Monitoring Function Protection. Figure 13 shows that when a module (named M) inside a VM attempts to modify the protected code and data (e.g., perform a modification operation to the system call table), an *ept* violation could be triggered into the KVM and the event will be captured by the protection module.

(3) Performance Testing. According to the actual experience, the ratio of random read and write operations inside an operating system is usually 7 : 3. Therefore, use the *Fio* test tool in a VM to perform the following operations to simulate normal read and write situations, and parameters of the test VM are OS is CentOS 7.2 x86_64, configured 2 vCPUs, and memory capacity is 4 GB.

"fio -ioengine = sync -bs = 4k -direct = 1 -thread -rw = randrw -rwmixread = 70 -size = 9G -filename = /home/ar/test -name = "test" -iodepth = 32 -runtime = 180"

At the same time, the following monitoring policy p is set for the VM, where the system call number indications are as follows: 0 indicates *sys_read*, 1 indicates *sys_write*, 2 indicates *sys_open*, and 3 indicates *sys_close*.

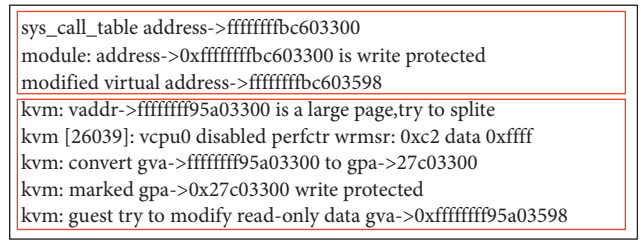

Figure 13: The protection result of the system call table of the VM was tempered by M.

Table 4: The statistics of test cases of IOPS (R: read, W: write, N: no monitoring, and Y: monitoring enable).

ID	IOPS				Bandwidth (kB/s)			
	R N	R Y	W N	W Y	R N	R Y	W N	W Y
1	71	69	30	29	285.6	279.9	120.7	118.3
2	71	70	30	29	284.9	281.6	120.4	119.0
3	72	69	30	29	288.8	279.1	122.4	117.9
4	71	70	30	29	286.7	283.6	121.2	119.7
5	71	70	30	29	287.9	281.6	121.7	119.1
6	71	70	30	29	284.4	281.7	120.2	119.1
7	71	70	30	29	284.6	280.7	120.2	118.7
8	71	70	30	29	286.4	281.5	121.1	118.9
9	71	70	30	29	287.1	281.1	121.4	118.9
10	71	70	30	29	286.1	283.9	121.09	119.9
Avg.	71.1	69.8	30	29	286.3	281.5	121.1	118.9

"p = (c9b6bea3-a2db-42f7-a625-1fbc8d601291, {0,1,2,3},(vim, /etc/hosts), record)"

In the case of random read/write ratio of 7 : 3, the statistical data of IOPS overhead brought by the monitoring mechanism, as shown in Table 4.

The results show that the monitoring system has little impact on VM IOPS.

For CPU and memory overhead brought by the monitoring mechanism, based on storage IOPS test case in the VM, the "*nmon*" test tool is used on the host to evaluate performance impact of the monitoring mechanism on the host system. The use cases and test data are shown in Tables 5 and 6. The test data shows that the performance consumption of CPU and memory brought by the monitoring mechanism is basically negligible.

Based on abovementioned discussion, the method is (1) compared with the internal monitoring mode because there is no need to install a visible agent inside the VM, and the method has higher reliability and security; (2) compared with the agentless monitoring mode based on VMI, the semantic gap between a VM and a VMM can be effectively eliminated by the monitoring code injected transparently. Moreover, resource consumed by the method is lower than the VMI method. (3) Compared with the method proposed in [1], the semantic analysis operation is performed every time when the monitoring is performed and the operation needs to be executed only once in the code injection phase by leveraging the proposed method, which greatly reduces the time-consuming switching of the context caused by the semantic analysis operation.

Table 5: Test cases of CPU and memory performance.

Test case	Test method description
Test 1: CPU performance overhead test	The host uses *nmon* to collect the CPU load once every 10 s and continuously collects 180 times for 30 minutes
Test 2: memory performance overhead test	The host uses *nmon* to collect the memory load once every 10 s and continuously collects 180 times for 30 minutes
PS: each test perform 4 tests (2 tests before monitoring function deployment and 2 tests after monitoring function deployment)	

Table 6: Test statistics of CPU and memory consumed.

Test items	No monitoring (%)	Monitoring enable (%)	Resource consumed (%)
CPU	2.67	2.7	0.03
Memory	17.477	18.112	0.635

5. Conclusions and Future Work

In this paper, the corresponding designs and techniques of IaaS security framework, security service access, network anomaly detection, and VM monitoring are designed and realized. The current research situation of cloud security framework, security service access, network anomaly detection, and VM monitoring are discussed, and advantages and disadvantages are pointed out. Many publicly available research results focus on abstracting the cloud security framework, which are not conducive to promotion and implementation of cloud security technologies rapidly. Besides, the detailed implementations and experimental campaign and discussion about the effectiveness and performance costs of the proposed framework and the various supporting securities are given. More complete technical discussions about the framework and concrete technologies (e.g., Encryption, VPN, VM escape detection, Data backup, and Software update) will be discussed in future work and a more comprehensive IaaS security protection framework, as shown in Figure 14, will be researched.

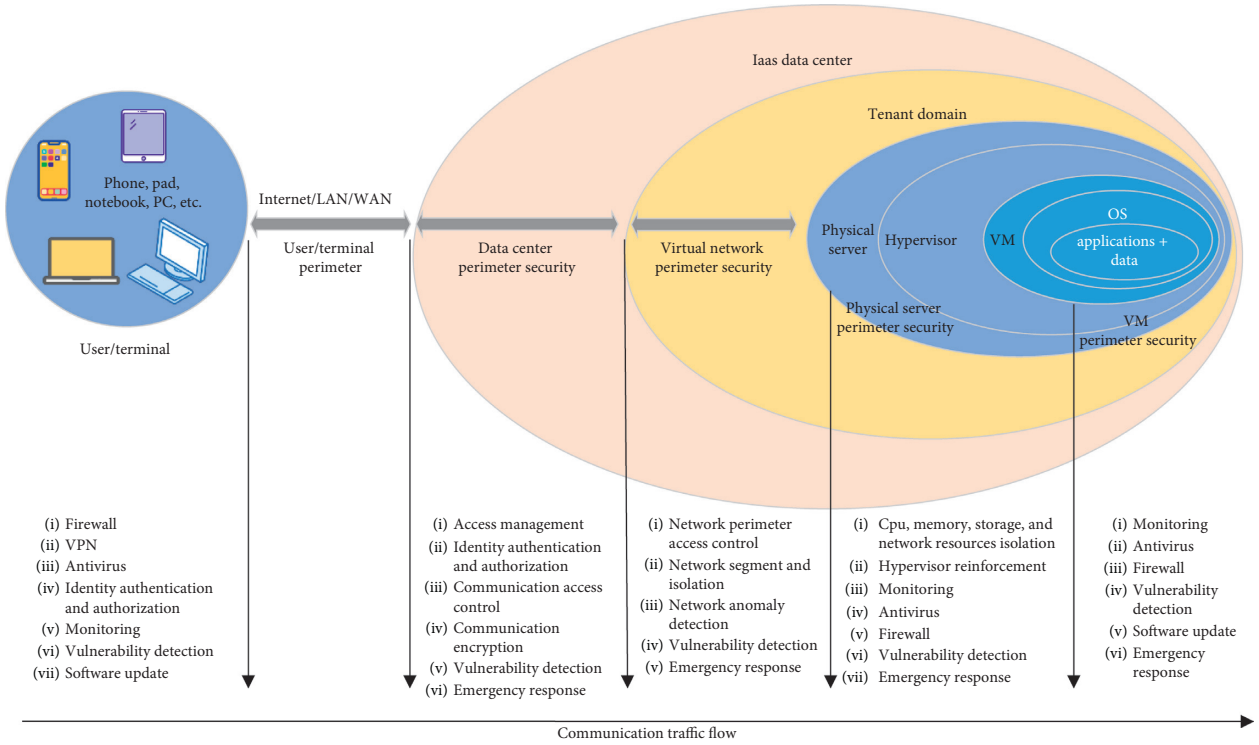

Figure 14: A more comprehensive security protection framework for a virtualized IaaS environment.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant nos. 61802270 and 61802271, Key Research and Development Project of Sichuan Province of China under Grant no. 2018GZ0100, and Fundamental Research Business Fee Basic Research Project of Central Universities under Grant no. 2017SCU11065.

References

[1] X. Y. Yin, X. S. Chen, L. Chen, G. L. Shao, H. Li, and S. Tao, "Research of security as a service for VMs in IaaS platform," *IEEE Access*, vol. 6, no. 1, pp. 29158–29172, 2018.

[2] G. Brunette and R. Mogull, *Security Guidance for Critical Areas of Focus in Cloud Computing v4.0*, Cloud Security Alliance, Toronto, Canada, 2017.

[3] CSA, *Trusted Cloud Initiative Reference Architecture v2.0*, Cloud Security Alliance, Toronto, Canada, 2013.

[4] NIST Cloud Computing Security Group, "NIST cloud computing security reference architecture," NIST Special Publication, Gaithersburg, MD, USA, 2011.

[5] ENISA, *Security Framework for Governmental Security Framework for Governmental Clouds*, ENISA, Heraklion, Greece, 2015.

[6] V. Varadharajan and U. Tupakula, "Security as a service model for cloud environment," *IEEE Transactions on Network and Service Management*, vol. 11, no. 1, pp. 60–75, 2014.

[7] C. Lin, W. B. Su, K. Meng, Q. Liu, and W. D. Liu, "Cloud computing security: architecture, mechanism and modeling," *Chinese Journal of Computers*, vol. 36, no. 9, pp. 1765–1784, 2013.

[8] VMware, *VMware NSX Network Virtualization Design Guide*, VMware, Palo Alto, CA, USA, 2013.

[9] Sugon, "Security vulnerability scanning system in cloud network environment," Sugon, Beijing, China, CN 103825891 A, 2014.

[10] IEEE, *IEEE 802.1BR—Bridge Port Extension*, IEEE LAN MAN Standards Committee, Piscataway, NJ, USA, 2012.

[11] IEEE, *IEEE 802.1Qbg—Edge Virtual Bridging*, IEEE LAN MAN Standards Committee, Piscataway, NJ, USA, 2012.

[12] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[13] A. H. Hamamoto, L. F. Carvalho, L. D. H. Sampaio, T. Abrão, and M. L. Proença, "Network anomaly detection system using genetic algorithm and fuzzy logic," *Expert Systems with Applications*, vol. 92, pp. 390–402, 2018.

[14] G. Fernandes Jr., L. F. Carvalho, J. J. P. C. Rodrigues, and M. L. Proença, "Network anomaly detection using IP flows with principal component analysis and ant colony optimization," *Journal of Network and Computer Applications*, vol. 64, pp. 1–11, 2016.

[15] C. Callegari, S. Giordano, and M. Pagano, "Entropy-based network anomaly detection," in *Proceedings of the 2017 International Conference on Computing, Net-working and*

Communications (ICNC), pp. 334–340, Santa Clara, CA, USA, January 2017.

[16] K. Flanagan, E. Fallon, P. Connolly, and A. Awad, "Network anomaly detection in time series using distance based outlier detection with cluster density analysis," in *Proceedings of the 2017 Internet Technologies and Applications (ITA)*, pp. 116–121, Wrexham, UK, September 2017.

[17] R. Chandramouli, "Security recommendations for hypervisor deployment," NIST Special Publication, Gaithersburg, MD, USA, 2015.

[18] X. D. Ren and J. Wang, "Linux system call hijacking detection method," *Information Security and Technology*, vol. 11, no. 6, pp. 61-62, 2015.

[19] I. Ahmed, A. Zoranic, S. Javaid, and G. G. Richard, "ModChecker: kernel module integrity checking in the cloud environment," in *Proceedings of the IEEE International Conference on Parallel Processing Workshops*, pp. 306–313, Pittsburgh, PA, USA, September 2012.

[20] J. Hizver and T. C. Chiueh, "Real-time deep virtual machine introspection and its applications," *ACM Sigplan Notices*, vol. 49, no. 7, pp. 3–14, 2014.

[21] W. J. Liu, L. N. Wang, C. Tan, and X. Lai, "A virtual machine introspection triggering mechanism based on VMFUNC," *Journal of Computer Research and Development*, vol. 54, no. 10, pp. 2310–2320, 2017.

[22] P. Malhotra, L. Vig, G. Shroff et al., "Long short term memory networks for anomaly detection in time series," in *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pp. 89–94, Bruges, Belgium, April 2015.