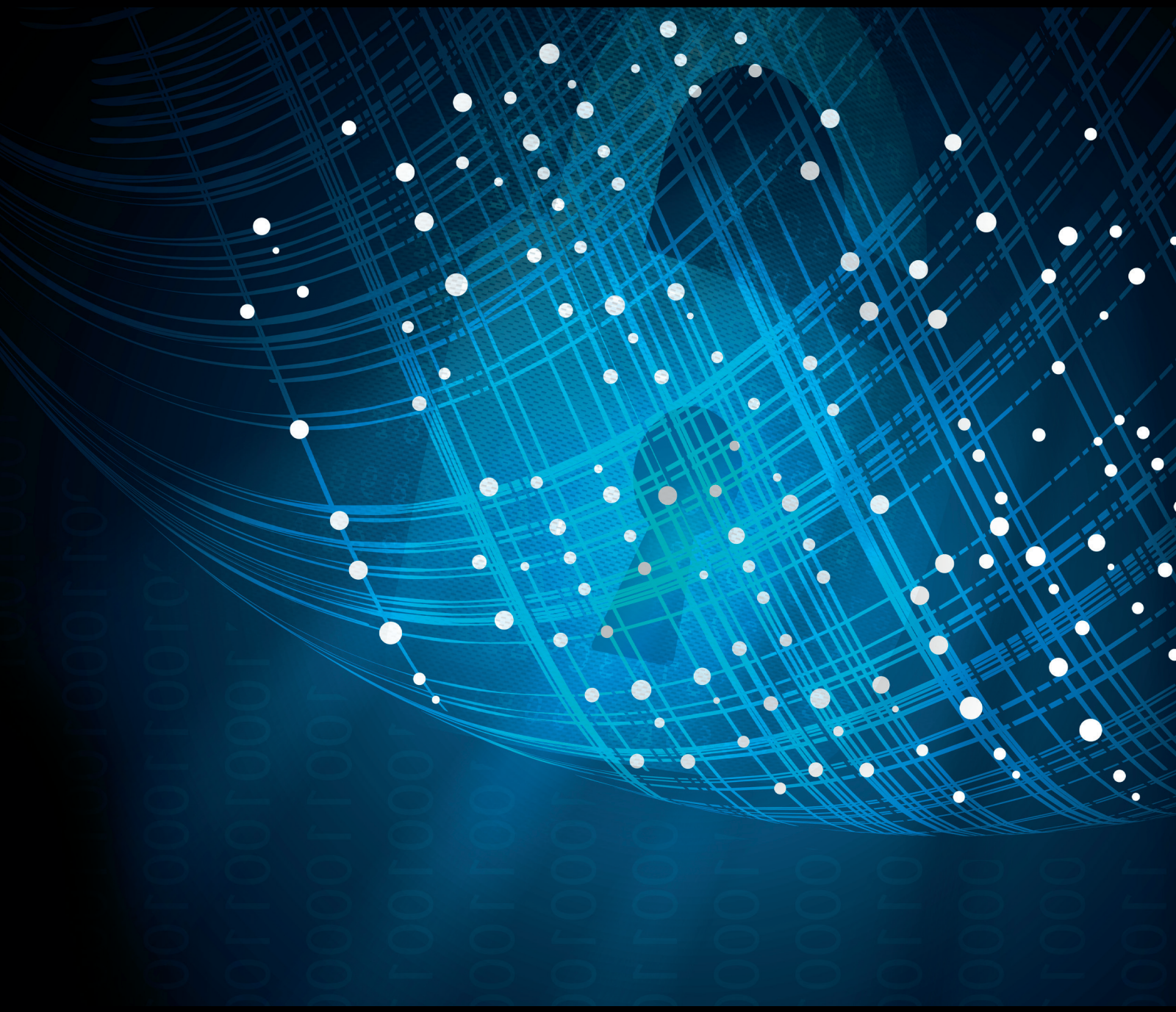
Security and Communication Networks

Network Security and Management in SDN

Lead Guest Editor: Zhiping Cai
Guest Editors: Chengchen Hu, Kai Zheng, Yang Xu, and Qiang Fu



Network Security and Management in SDN

Security and Communication Networks

Network Security and Management in SDN

Lead Guest Editor: Zhiping Cai
Guest Editors: Chengchen Hu, Kai Zheng, Yang Xu, and Qiang Fu

Editorial Board

Contents

Hindawi
Security and Communication Networks
Volume 2018, Article ID 7928503, 2 pages
https://doi.org/10.1155/2018/7928503

Editorial

Network Security and Management in SDN

Zhiping Cai,[1] Chengchen Hu,[2] Kai Zheng,[3] Yang Xu,[4] and Qiang Fu[5]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, China*
[2]*Xi'an Jiaotong University, Xi'an 710049, China*
[3]*Huawei Technologies, Shenzhen 518129, China*
[4]*New York University, New York City, NY 11201, USA*
[5]*Victoria University of Wellington, Wellington 6140, New Zealand*

Correspondence should be addressed to Zhiping Cai; zpcai@nudt.edu.cn

Received 17 February 2018; Accepted 20 February 2018; Published 4 June 2018

Software Defined Networking (SDN) enables flexible deployment and innovation of new networking applications by decoupling and abstracting the control and data planes. It has radically changed the concept and way that we build and manage networked systems and reduced the barriers to entry for new players in the service market. Recently, SDN has been widely studied and applied to facilitate network management and the development of network security systems. However, the separation of control and data planes makes SDN vulnerable to security threats. Attackers can monitor and tamper network management information and disrupt network communication by implementing man-in-the-middle attacks, saturation attacks, Denial of Service (DoS) attacks, and so forth. Therefore, it is important to analyze the vulnerability and design defense mechanisms for securing SDN-based systems. In this special issue, we have selected nine papers that address such technical issues.

B. Han et al. propose a cross-plane distributed DoS (DDoS) attack defense framework in SDN, called OverWatch, which exploits collaborative intelligence between data plane and control plane with high defense efficiency. They develop a collaborative DDoS attack detection mechanism, which consists of a coarse-grained flow monitoring algorithm on the data plane and a fine-grained machine learning based attack classification algorithm on the control plane. J. Ye et al. apply the support vector machine classification algorithm to judge the network traffic and detect the DDoS attack. T. Wang et al. also try to address the DoS attack problem. They propose a lightweight and fast DoS detection and mitigation system for SDN, called SDNManager. The SDNManager employs a novel dynamic time-series model which greatly improves bandwidth prediction accuracy. They also propose a dynamic controller scheduling strategy to ensure the global network state optimization and improve the defense efficiency.

Y. Zhou et al. discovered a novel inference attack targeted at SDN/OpenFlow network, which is motivated by the limited flow table capacities of SDN/OpenFlow switches and the following measurable network performance decrease resulting from frequent interactions between data and control plane when the flow table is full. They also propose two possible defense strategies for the discovered vulnerability, including a routing aggregation algorithm and a multilevel flow table architecture. C. Qi et al. present a game-theoretic model to analyze the security performance of SDN architectures. This model can represent several kinds of player information, simulate approximate attack scenarios, and quantitatively estimate systems' reliability. Their experimental results and analysis reveal diverse defense mechanisms adopted in dynamic systems, which have different effects on security improvement.

W. Fu et al. analyze the forwarding procedure and identify the performance bottleneck of SDN software switches. An FPGA-based mechanism for accelerating and securing SDN switches, named FAS, is proposed to take advantage of the reconfigurability and high-performance advantages of FPGA. FAS improves the performance as well as the capacity against malicious traffic attacks of SDN software switches by offloading some functional modules. Y. Lee et al. propose

Duo, an intrusion tolerant system in SDN, which can reduce exposure time without consuming computing resources. Duo classifies traffic into benign and suspicious traffic with the help of SDN/NFV technology that also allows dynamically forwarding the classified traffic to different servers. By reducing exposure time of a set of servers, Duo can decrease exposure time on average.

C. Zhang et al. propose Kuijia, a robust traffic engineering system for data center WANs, which relies on a novel failover mechanism in the data plane called rate rescaling. The victim flows on failed tunnels are rescaled to the remaining tunnels and put in lower priority queues to avoid performance impairment of aboriginal flows. Real system experiments show that Kuijia is effective in handling network faults and significantly outperforms the conventional rescaling method. Y. Shi et al. propose CHAOS, an SDN-based moving target defense system. A Chaos Tower Obfuscation (CTO) method is proposed to depict the hierarchy of all the hosts in an intranet and define expected connections and unexpected connections. Moreover, they develop fast CTO algorithms to achieve a different degree of obfuscation for the hosts in each layer. The proposed approach makes it very easy to realize moving target defense in networks.

Zhiping Cai
Chengchen Hu
Kai Zheng
Yang Xu
Qiang Fu

Hindawi
Security and Communication Networks
Volume 2018, Article ID 9804061, 8 pages
https://doi.org/10.1155/2018/9804061

Research Article

A DDoS Attack Detection Method Based on SVM in Software Defined Network

Jin Ye,[1,2] Xiangyang Cheng,[1,2] Jian Zhu,[1,2] Luting Feng,[1,2] and Ling Song[1,2]

[1]*School of Computer and Electronic Information, Guangxi University, Nanning 530004, China*
[2]*Guangxi Key Laboratory of Multimedia Communications and Network Technology, Nanning 530004, China*

Correspondence should be addressed to Ling Song; aling7197_cn@sina.com

Received 13 October 2017; Revised 28 December 2017; Accepted 24 January 2018; Published 24 April 2018

Academic Editor: Zhiping Cai

The detection of DDoS attacks is an important topic in the field of network security. The occurrence of software defined network (SDN) (Zhang et al., 2018) brings up some novel methods to this topic in which some deep learning algorithm is adopted to model the attack behavior based on collecting from the SDN controller. However, the existing methods such as neural network algorithm are not practical enough to be applied. In this paper, the SDN environment by mininet and floodlight (Ning et al., 2014) simulation platform is constructed, 6-tuple characteristic values of the switch flow table is extracted, and then DDoS attack model is built by combining the SVM classification algorithms. The experiments show that average accuracy rate of our method is 95.24% with a small amount of flow collecting. Our work is of good value for the detection of DDoS attack in SDN.

1. Introduction

With the continuous development of network technology, the ceaseless expansion of network business needs, and rapid growth of the Internet economy in the Internet age, the services of network with important business and industry information have been spread to the production and life of current society. The emergence of DDoS attacks can lead to abnormalities in the related network services, causing huge economic losses and even causing other catastrophic consequences. DDoS attacks are one of the serious network security threats facing the Internet. It is a key research topic in the security field to detect DDoS attacks accurately and quickly. SDN is an emerging network innovation architecture that separates the network data plane and the control plane [1, 2], which has the characteristics of network programmable, centralized management control, and interface opening.

Network attackers attack network bandwidth, system resources, and application resources, to achieve the effect of denial of service attacks. DDoS attacks show the increasing scale of attack; the attack mode is more intelligent. The difficulties of DDoS attack detection are as follows: (1) the attack traffic characteristics not being easy to identify; (2) the lack of collaboration between the coherent network nodes; (3) the change of the attack tool being strengthened, with the threshold of its use decreasing; (4) the widely used address fraud making it difficult to trace the source of the attack; (5) the duration time of attack being short and response time being limited.

In the traditional network architecture, the main methods of DDoS attack detection technology can be divided into attack detection based on traffic characteristics and attack detection based on traffic anomaly. The former mainly collects all kinds of characteristics information related to the attack and establishes a characteristics database of DDoS attack. By comparing and analyzing the data information of the current network data packet and characteristics database, we can judge whether it is attacked by DDoS or not. The main implementation methods are characteristics match, model reasoning, state transition, and expert systems. The latter is mainly to establish traffic model and analysis of abnormal flow changes, to determine whether the traffic is abnormal or not, so as to detect whether the server was attacked.

Under the innovative architecture environment of SDN, deep packet analysis is available through the full network view [3, 4]. It supports quick response and update of traffic policies and rules. The SDN has the capability of perceived control of the global visualization view, flexible

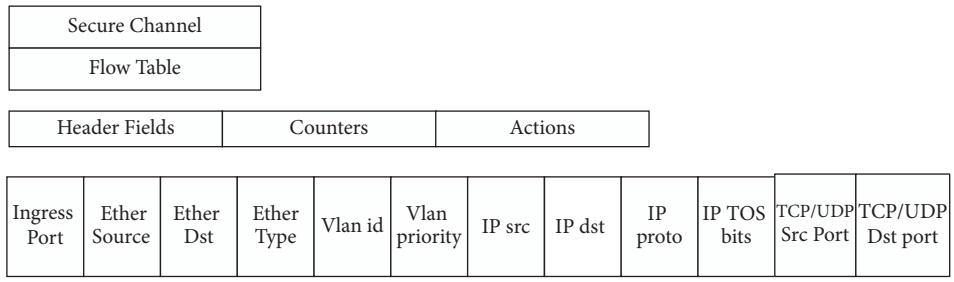


Figure 1: Flow table structure.

and schedulable rapid deployment capability, and service open intelligent scheduling capability. While ensuring network services and reducing deployment costs, the software defined network enhances the quality of user experience and facilitates the promotion of the whole network deployment.

Researchers aimed at traditional network architecture proposed a lot of DDoS attack detection methods. Lin and Wang [5] proposed a DDoS attack detection and defense mechanism based on SDN, but the method used three Openflow management tools with sFlow standard to perform anomaly detection, so the deployment and operation are complex. Yang et al. [6] dished a method in which the flow information and the IP entropy characteristic information are combined, which is detected by a single flow information and IP entropy characteristic information, which has a higher and more accurate detection effect. Although information entropy is flexible and convenient, it still needs to be combined with other technologies in determining the threshold and multielement weight distribution. Saied et al. [7] advanced that based on analysis the characteristics of each protocol of TCP/UDP/ICMP through the training ANN algorithm to detect DDoS attacks, the method needs to distinguish packet protocol, which is complex and inefficient.

In [8], the SOM algorithm is used to detect DDoS attacks by extracting the flow statistics related to DDoS attacks. This method has the characteristics of low consumption and high detection rate. The key point lies in the extraction of time interval. The disadvantage of this method is that the detection has a certain hysteresis and the attack behavior is not timely and accurately found. In [9], the authors proposed a framework for detection and mitigation of DDoS attacks in a large-scale network, but it is not suitable for small-scale deployment. In [10], a DDoS attack detection mechanism based on a legitimate source and destination IP address database is proposed. Based on the nonparametric cumulative algorithm CUSUM, it analyzes the abnormal characteristics of the source IP address and the destination IP address when the DDoS attack occurs and effectively checks the DDoS attack, but the method needs to adjust and determine the threshold.

It is concluded that DDoS attack detection in SDN networks mainly includes information entropy and utilization of data mining algorithm, in which the more popular is the SOM algorithm. Due to the high false positive rate of information entropy, the SOM algorithm needs to determine the number of neurons in advance. Therefore, in this paper, we summarize the characteristics of several DDoS attacks, then collect the switch flow table information, extract the six-tuple characteristic values matrix, and establish their SVM classification model. The algorithm can process multidimensional data and map the low-dimensional nonlinear separable data into the high-dimensional feature space to make it linearly separable and able to be classified with high accuracy. At present, the algorithm is widely used in anomaly detection and classification.

This paper is organized as follows: Section 1 describes the introduction; Section 2 gives a detailed description of the SVM classification model; Section 3 illustrates the experimental method presented in this paper; Section 4 summarizes the paper.

2. DDoS Detection Based on Support Vector Machine (SVM)

In the SDN architecture, the Openflow switch forwards the main network data at a high speed [11]. The SDN controller is responsible for the forwarding and management of the forwarding decision and the collection of traffic information of switches. In the SDN switch, the core data structure of the forwarding policy management control is the flow table [12]. The SDN manages the relevant network traffic by searching the flow table entries, where the flow entry can forward the packet to one or more interfaces. Each entry includes the header field, the counters, and the actions. The packet forwarding of the switch is based on the flow table. Each flow table is composed of multiple flow entries. The flow table entries form the rules for data forwarding. Figure 1 shows the flow table entry structure diagram.

The flow diagram of the attack detection consists mainly of the flow state collection, the extraction characteristic values, and the classifier judgment, as shown in Figure 2. The flow state collection periodically sends a flow table request to the Openflow switch and sends the flow table information replied from the switch to the flow state collection. The characteristic values extraction is mainly responsible for extracting the characteristic values related to the DDoS attack from the switch flow table and composing the six-tuple characteristic values matrix. Six-tuple characteristic values information is classified by using an SVM-based algorithm [13] to distinguish between normal traffic and attacking abnormal traffic.

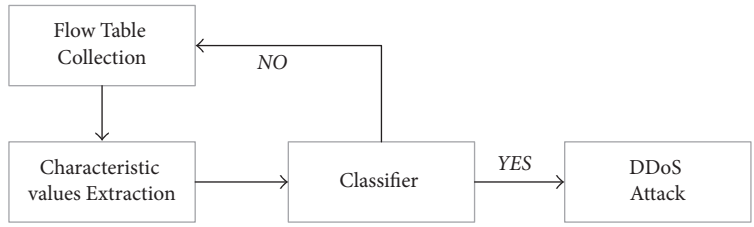


Figure 2: Attack detection process.

2.1. Flow Status Collection. In the SDN network environment, the collection of the flow table status information is mainly accomplished through the Openflow protocol. The switch responds to the *onp_flow_stats_request* message periodically sent by the controller, and the time interval between getting the flow tables should be moderate, setting the flow table obtaining period to be consistent with the flow deleting time set by the floodlight controller and running the "sudo ovs-ofct1 dump-flows s1" command to collect the status information of the flow table. The flow table information extracted by the switch is given as follows:

NXST_FLOWreply($x$id = 0 × 4) : cookie = 0 × 0, duration = 21.098 s, table = 0, n_packets = 1, n_bytes = 42, idle_timeout = 60, idle_age = 21, priority = 65535, arp, in_port = 2, vlan_tci = 0 × 0000, dl_src = $c6 : 76 : 11 : 0a : 4c$: 78, dl_dst = $82 : 0d : bf : d2 : ad : f0$, arp_spa = 10.0.0.3, arp_tpa = 10.0.0.1, arp_op = 1actions = output : 1.

2.2. Extract the Characteristic Values. When DDoS attack occurs on the network, for it is controlled by the program, the network will randomly forge a large number of source IP addresses to send a certain size of the packet to attack the target. In the network, the attack flow shows certain similarity, regularity, and then it can be detected by analyzing the characteristic values information of the flow table. In [14], the author does not mention the change of the speed of source port in attack detection when extracting the traffic characteristic values, and a large number of new port addresses were randomly generated in the attack process.

In this paper, some existing research on SDN is analyzed and compared and the data analysis and processing are carried out by extracting the flow status information on the basis of previous research. The following six-tuple characteristic values related to DDoS attacks are obtained for DDoS attack detection.

(1) The speed of source IP (SSIP) is the number of source IP addresses per unit of time:

$$\text{SSIP} = \frac{\text{Sum\_IP}_{\text{src}}}{T}, \tag{1}$$

where $\text{Sum\_IP}_{\text{src}}$ is the source IP number and T is the sampling interval. In the event of an attack, a large number of attacks are generated by random forgery to send data packets, the source IP address number will increase rapidly.

(2) The speed of source port (SSP) is the number of source ports per unit of time

$$\text{SSP} = \frac{\text{Sum\_port}_{\text{src}}}{T}, \tag{2}$$

where $\text{Sum\_port}_{\text{src}}$ is the number of attack source ports. When a large number of attack requests occur, a large number of port numbers are randomly generated.

(3) The Standard Deviation of Flow Packets (SDFP), that is, the standard deviation of the number of packets in the T period, is as follows:

$$\text{SDFP} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\text{packets}_i - \text{Mean\_packets}\right)^2}, \tag{3}$$

where $\text{Mean\_packets} = (1/N)\sum_{i=1}^{N}\text{packets}_i$ represent the average number of the packets in the T period. N is the total number of flow entries per period, in the event of an attack; in order to produce the attack effect, the general attack data packets are relatively small and the standard deviation of flow packets will be smaller than the normal flow.

(4) The Deviation of Flow Bytes (SDFB), that is, the standard deviation of the number of bits in the T period, is as follows:

$$\text{SDFB} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\text{bytes}_i - \text{Mean\_bytes}\right)^2}, \tag{4}$$

where $\text{Mean\_bytes} = (1/N)\sum_{i=1}^{N}\text{bytes}_i$, represent the average of the number of bits in the T period. In the event of an attack, in order to reduce the packet load, attacker will send a smaller bit of data packets and the standard deviation flow bits will be smaller than the normal flow.

(5) The speed of flow entries (SFE), that is, the number of flow entries per unit time, is as follows:

$$\text{SFE} = \frac{N}{T}. \tag{5}$$

In the event of an attack, the number of flow entries per unit time increases dramatically, significantly higher than the normal value.

(6) The Ratio of Pair-Flow (RPF), that is, the ratio of interactive flow entries to total flow entries, is as follows:

$$\text{RPF} = \frac{2 * \text{Pair\_Sum}}{N}, \tag{6}$$

where Pair_Sum is the number of interactive flow entries. Under normal circumstances, the source host sends a request to the destination host to generate an interactive flow, which constitutes the following conditions.

The source IP of packet_i is the same as the destination IP of packet_j. The destination port number of packet_i is the same as the source port number of packet_j. The destination IP of packet_i is the same as the source IP of packet_j, and the source port number of packet_i is the same as the destination port number of packet_j. There will be two interactive flow entries in the flow table that satisfy Formula (7)

$$\begin{aligned} \text{Src\_IP}_i &= \text{Dst\_IP}_j, \\ \text{Src\_port}_i &= \text{Dst\_port}_j, \\ \text{Src\_IP}_j &= \text{Dst\_IP}_i, \\ \text{Dst\_port}_j &= \text{Src\_port}_i. \end{aligned} \tag{7}$$

When an attack occurs, the flow entries sent to the destination host in a T period increase sharply, the destination host cannot respond to the interactive flow in time, and in genera the attacker typically uses massive pseudosource addresses when attacking, so the number of interactive flow entries per will drop in the T period.

2.3. Classifier Judgment. We can think of attack detection as a classification problem, that is, classifying the given data and judging that whether the current network state is normal or abnormal. In the classifier judgment, the extracted six-tuple characteristic values are used for classification learning to determine whether the traffic is abnormal. Attack detection of the basic process is as follows: the network data is extracted as a six-tuple characteristic values sequence according to the time interval, and the sample sequence is given a {normal, abnormal} flag, which represents the two states of the network.

The appropriate machine learning algorithm is selected to construct the detection model according to the sequence of characteristic values samples and the unlabeled characteristic values samples are classified by using the model. This paper chooses a classification learning method based on support vector machine (SVM) algorithm [13, 15]. SVM is a learning method based on statistical learning theory. It can get good classification results without a lot of training data. It maps the nonlinearly separable sample set to a high-dimensional or even infinite dimensional feature space to make it linearly separable and find the optimal classification surface in this high-dimensional feature space. The kernel function in SVM effectively solves the problem of dimensionality disaster caused by high-dimensional mappings and enhances the ability of processing high dimension small sample data.

SVM is applied to DDoS attack detection with good accuracy. The DDoS attack detection method proposed in this paper uses a supervised learning algorithm. Firstly, flow table entries in the switch are sampled at a time interval T, and the characteristic values of the flow table entries in each sampling are calculated to obtain a sample set Z, which is expressed as $Z = (X, Y)$, where X represents flow table entries six-tuple characteristic values matrix, Y is the category marker vector corresponding to X: "0" represents normal state, and "1" represents attacked state. In the experiment, we attacked during $T20$–$T40$ periods. We marked the corresponding class labelled "1," and the remaining class labels were all "0" and then used the SVM classifier to train the sample set to obtain its parameters. Finally, we use trained SVM model to classify the unlabeled samples. If there is a sample marked "1," it is considered that an attack was made during the corresponding detection period.

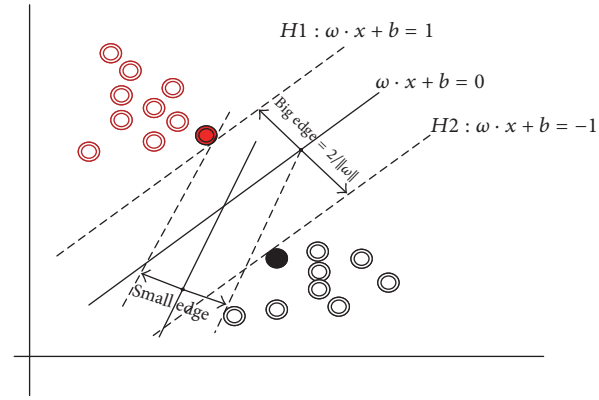


Figure 3: Classification hyperplane.

2.4. SVM. SVM is derived from the linearly separable optimal classification hyperplane, and its basic idea can be explained by the two-dimensional case of Figure 3. There is a training set $D = \{(X_1, y_1), (X_2, y_2), \ldots, (X_n, y_n)\}$, where X_i is the characteristic vector of the training sample and y_i is the associated class label. y_i takes +1 or −1 ($y_i \in \{+1, -1\}$, in this experiment, and y_i takes 1 or 0), indicating that the vector belongs to this class or not. It is said to be linearly separable if there is a linear function that can completely separate the two classes; otherwise it is nonlinearly separable.

Figure 3 is a linear separable case, since a straight line can be drawn to separate the vector of class +1 from the vector of class −1. There are countless such lines, and the so-called optimal classification line requires that the two samples be correctly separated and that the separation interval be the largest. SVM completes the classification of the sample by searching for the one that has the largest classification interval. The optimal classification line can be expressed by the equation $\omega \cdot x + b = 0$ ($\omega \in R^n$, $b \in R$); ω is the weight vector and b is the scalar, called the bias. The points above the separation hyperplane are satisfied

$$\omega \cdot x + b > 0. \tag{8}$$

Similarly, the points below the separation hyperplane are satisfied

$$\omega \cdot x + b < 0; \tag{9}$$

we can adjust the weight to make the edge side of the hyperplane able to be expressed as

$$H1 : \omega \cdot x + b \geq 1, \quad \text{for } y_i = 1$$

$$H2 : \omega \cdot x + b \leq 1, \quad \text{for } y_i = -1. \tag{10}$$

This means that the vectors falling on or above $H1$ belong to class +1 and the vectors falling on or below $H2$ belong to −1. From (10) we can get

$$y_i (\omega \cdot x + b) \geq 1, \quad \forall i. \tag{11}$$

Any of the training tuples falling on $H1$ and $H2$ are support vectors, and the equal sign is established.

From the above, we can get that the maximum edge is $2/\|\omega\|$. Finding that the maximum value of $2/\|\omega\|$ is equivalent to calculating the minimum value of $\|\omega\|$. Generalized to n-dimensional space, how the SVM finds the optimal hyperplane is equivalent to solving the constrained optimization problem; the formula is expressed as

$$\begin{aligned} &\min_{w,b} \quad \frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{N}\xi_i \\ &\text{s.t.} \quad y_i (\omega \cdot x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \ i = 1, \ldots, N, \end{aligned} \tag{12}$$

where $C > 0$ is the penalty parameter, indicating the degree of attention to the outliers, and the relaxation variable ξ_i is a measure of the degree of outliers [16].

DDoS attack detection is equivalent to two-classification problem; we use the SVM algorithm characteristics, collect switch data to extract the characteristic values to train, find the optimal classification hyperplane between the normal data and DDoS attack data, and then use the test data to test our model and get the classification results.

3. Experiment and Analysis

In this experiment, the controller (Floodlight [17]) and the switch (Openflow switch) are deployed under Ubuntu to generate the network topology diagram in Figure 4. The experimental topology is generated by mininet. The validity of DDoS attack detection method is verified by deploying SDN environment. PC1 and PC2 are the bot hosts; PC5 is the victim target. PC1 and PC2 can send normal packets to generate normal samples or send DDoS attack packets to generate DDoS attack samples. PC3 and PC4 generate normal network traffic samples. These samples are used for training to generate model and detecting attack.

During the training sample phase, the normal traffic is generated by PC3 and PC4. It includes TCP traffic, UDP traffic, and ICMP traffic. We use the classic DDoS attack tool Hping3 to generate abnormal network traffic. Hping3 is fully scriptable using the TCL language and can receive and send data packets by describing the binary or string representation of the data packets. In practice this means that a few lines of code can perform things that usually take many lines of C code. Examples are automated security tests with pretty printed report generation, TCP/IP test suites, many kind of attacks, NAT-ting, prototypes of firewalls, implementation of routing protocols, and so on. The advantage of hping3 is the ability to customize parts of the packet, so users can

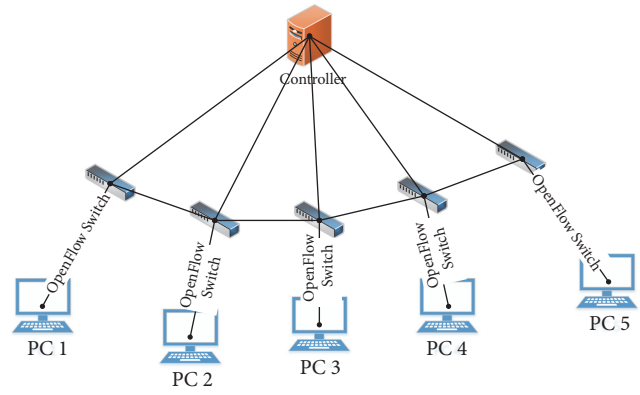


Figure 4: Network topology.

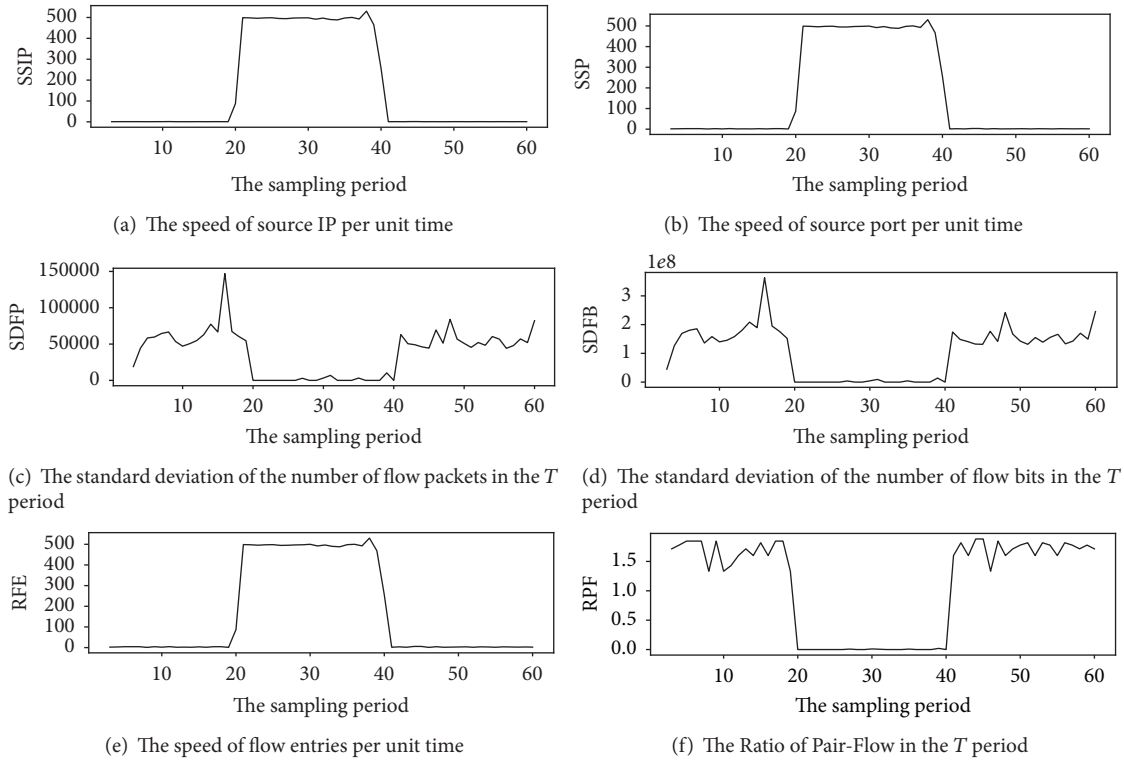
Table 1: The training and detection of attack flow samples.

Attack types	Training	Detection
TCP(200) flood		>30000
TCP(600) flood		>30000
TCP(1000) flood		>30000
UDP(200) flood		>30000
UDP(600) flood	>30000	>30000
UDP(1000) flood		>30000
ICMP(200) flood		>30000
ICMP(600) flood		>30000
ICMP(1000) flood		>30000

flexibly attack and detect the target [18]. Based on the above characteristics, we use Hping3 to generate different types of attack data. We use it to simulate the typical network traffic attack TCP SYN flood, UDP flood, and ICMP flood. These floods are used as training and for detection of attack samples. The types of attacks and the number of flows are shown in Table 1. The numbers in brackets are the size of the packets at the time of attack. They are same as the size of the packets of training data. We use the training data to generate the model. The training model is used to detect different attack data.

In this experiment, the sampling period T (interval) is 3 s. We attack in the $T20$ to $T40$ periods. During the sampling process, we collect the flow table data of 60 periods in the Openflow switch, then process and normalize the data of each period, and get the normal samples and DDoS attack flow samples of the six-tuple characteristic values matrix. The trends of the six-tuple characteristic values in 60 periods are shown in Figure 5.

In Figure 5, the abscissa represents period and the ordinate indicates the speed of source IP in a unit time (Figure 5(a)), the speed of source port in a unit time (Figure 5(b)), the standard deviation of the number of flow packets in the T period (Figure 5(c)), the standard deviation of the number of flow bits in the T period (Figure 5(d)), the speed of flow entries in a unit time (Figure 5(e)), and the Ratio of Pair-Flow in a T period (Figure 5(f)). In the experiment, we attack the $T20$–$T40$ periods. In the event of an attack, the number of flow entries per unit time will increase dramatically.



(a) The speed of source IP per unit time

(b) The speed of source port per unit time

(c) The standard deviation of the number of flow packets in the T period

(d) The standard deviation of the number of flow bits in the T period

(e) The speed of flow entries per unit time

(f) The Ratio of Pair-Flow in the T period

Figure 5: Six-tuple eigenvalue trend.

Generally, the attack is based on the pseudosource random IP addresses and port numbers. The amount of source IP and the number of source ports are also increased in a unit time. So there are similar growth trends in Figures 5(a), 5(b), and 5(e). Under normal circumstances, sending the data packets is relatively large, and in the attack, in order to achieve the attack effect, attacker usually sends data as soon as possible, so the data packets are relatively small and unchanged. Thus, the standard deviations of the number of flow packets and the number of flow bits in a T period are relatively small and have tiny fluctuations. As shown in Figures 5(c) and 5(d), the two characteristic parameters are large and fluctuating obviously in the normal periods, and they are very small and change gently in the $T20$–$T40$ periods. When we access the network normally, the source host and the destination host will produce interactive flow entries. In the time of an attack, due to using virtual random source IP addresses and source port numbers commonly, when the large amount of requests occur, the destination host cannot respond timely. Therefore, the proportion of interactive flow will decrease sharply. As shown in Figure 5(f), in the $T20$–$T40$ periods, the interactive flow entries drop to almost zero. Under the normal circumstances, the ratio of interactive flow entries is relatively large and fluctuates in a normal range.

We used the SVM function in Rstudio [19] to train the data to get the SVM model and use the model to predict the test data. We use the two characteristic values SSIP and RPF in the test data to draw classification chart; the classification results are shown in Figure 6.

In the experiment, the experimental data is nonlinear separable, and it is multidimensional, so the classification hyperplane is not a straight line or a plane but a curved surface (two-dimensional image displays curve). The light green area is the normal network access data. The pink area indicates that the network is being attacked. The red marks are the data distribution of the network being attacked. "×" represents the support vectors in this figure.

The performance of the attack detection is displayed by the detection rate (DR) and false alarm rate (FAR); the formulas are calculated as the values:

$$\mathrm{DR} = \frac{\mathrm{DD}}{\mathrm{DD} + \mathrm{DN}}. \tag{13}$$

In this formula, DD indicates that the attack flow is detected as an attack flow, and DN means that the attack flow is detected as a normal flow.

$$\mathrm{FAR} = \frac{\mathrm{FD}}{\mathrm{FD} + \mathrm{TN}}. \tag{14}$$

In the formula, FD means that the normal flow is detected as an attack flow, and TN indicates that the normal flow is detected as a normal flow.

In the experiment, the normal traffic is composed of three basic communication kinds of traffic (TCP, UDP, and ICMP) and the attack traffic consists of three separate types of attack traffic: TCP, UDP, and ICMP. The accuracy rate and false alarm rate of packet detection for different lengths of the three types of attack traffic are shown in Table 2. The average detection accuracy rate of this experiment is 95.24%, and the average false alarm rate is 1.26%, and the expected effect was achieved. The low false alarm rate is a good result and, on the other hand, it may be that our simulation of normal

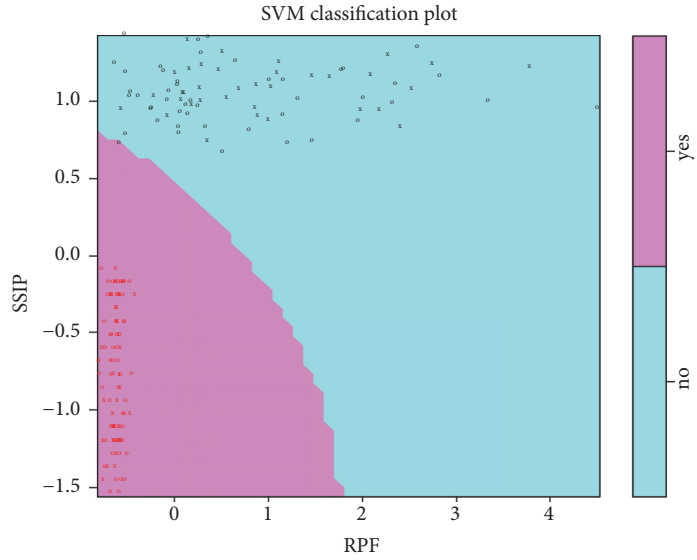


Figure 6: Classification results.

Table 2: The experimental results of three kinds of attacks.

	TCP			UDP			ICMP		
Packets size	200	600	1000	200	600	1000	200	600	1000
Detection accuracy rate	95.24%	100%	95.24%	95.24%	95.24%	95.24%	90.48%	95.24%	95.24%
Average		96.83%			95.24%			93.65%	
Average detection accuracy rate					95.24%				
False alarm rate	0.0%	0.0%	0.0%	2.7%	0.0%	0.0%	5.88%	0.0%	2.77%
Average		0.0%			0.9%			2.88%	
Average false alarm rate					1.26%				

data flow is not comprehensive enough, which is what we need to improve in the future. The relatively low accuracy rate of ICMP flow detection may be due to the fact that the ICMP traffic has no source port and destination port, so the characteristic matrix is only 4 dimensions. But our experimental results still have a high detection accuracy rate, which reached our goal.

4. Concluding Remarks

In this paper, the flow status information of the network traffic is collected on the switch by the controller. We extracted the six-tuple characteristic values related to DDoS attack and then use the support vector machine algorithm to judge the traffic and carry out DDoS attack detection. We focus on the analysis of the changes of the characteristic values of traffic and verify the feasibility of this method by deploying the SDN experimental environment. The detection accuracy rate of the experiment is high and the false alarm rate is low, which has obtained our expected results. In comparison, the test detection accuracy rate of ICMP attack flow is relatively low. By analyzing the ICMP traffic, we have come to the conclusion that the ICMP flow has no source port and destination port, so SSP and RPF are zero, which makes the six-tuple characteristic values matrix change into four-tuple characteristic values matrix, whether attacked or not. But this has little effect on the experimental results, and our experiment has achieved the goal. On the other hand, due to the very low false alarm rate, we should simulate the normal data flow more comprehensively, which is what we need to improve in the future.

Conflicts of Interest

There are no conflicts of interest in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (nos. 61762030, 61462007).

References

[1] H. Zhang, Z. Cai, Q. Liu, Q. Xiao, Y. Li, and C. F. Cheang, "A surveyon security-aware network measurement in SDN," *Security and Communication Networks*, Article ID 2459154, 2018.

[2] J. Cao, M. Xu, Q. Li, K. Sun, Y. Yang, and J. Zheng, "Disrupting SDN via the data plane: a low-rate flow table overow attack," in *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks*, Niagara Falls, Canada, October 2017.

[3] Z. Cai, Z. Wang, K. Zheng, and J. Cao, "A distributed TCAM coprocessor architecture for integrated longest prefix matching, policy filtering, and content filtering," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 417–427, 2013.

[4] Y. Li, Z. Cai, and H. Xu, "LLMP: exploiting LLDP for latency measurement in software-defined data center networks," *Journal of Computer Science and Technology*, vol. 33, no. 2, pp. 277–285, 2018.

[5] H. Lin and P. Wang, "Implementation of an SDN-based security defense mechanism against DDoS attacks," in *Proceedings of the 2016 Joint International Conference on Economics and Management Engineering (ICEME 2016) and International Conference on Economics and Business Management (EBM 2016)*, Pennsylvania, Penn, USA, 2016.

[6] J. G. Yang, X. T. Wang, and L. Q. Liu, "Based on traffic and IP entropy characteristics of DDoS attack detection method," *Application Research of Computers*, vol. 33, no. 4, pp. 1145–1149, 2016.

[7] A. Saied, R. E. Overill, and T. Radzik, "Detection of known and unknown DDoS attacks using artificial neural networks," *Neurocomputing*, vol. 172, pp. 385–393, 2016.

[8] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks (LCN '10)*, pp. 408–415, Denver, Colo, USA, October 2010.

[9] N. Z. Bawany, J. A. Shamsi, and K. Salah, "DDoS attack detection and mitigation using SDN: methods, practices, and solutions," *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 425–441, 2017.

[10] X. Wang, M. Chen, C. Xing, and T. Zhang, "Defending DDoS attacks in software-defined networking based on legitimate source and destination IP address database," *IEICE Transaction on Information and Systems*, vol. E99D, no. 4, pp. 850–859, 2016.

[11] J. Xia, Z. Cai, G. Hu, and M. Xu, "An active defense solution for ARP Spoo ng in OpenFlow network," *Chinese Journal of Electronics*, vol. 3, 2018.

[12] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proceedings of the 2015 International Conference on Computing, Networking and Communications, ICNC 2015*, pp. 77–81, Garden Grove, Calif, USA, February 2015.

[13] M. Alazab, "Profiling and classifying the behavior of malicious codes," *The Journal of Systems and Software*, vol. 100, pp. 91–102, 2015.

[14] H. F. Li, X. L. Huang, and Z. Q. Zheng, "DDoS attack detection method based on software definition network and its application," *Computer Engineering*, vol. 42, no. 2, pp. 118–123, 2016.

[15] X. Nguyen, L. Huang, and A. D. Joseph, *Machine Learning and Knowledge Discovery in Databases*, Springer, Berlin, Germany, 2008.

[16] W. U. Shao-Hua, S. B. Cheng, and H. U. Yong, "Web attack detection method based on support vector machines," *Computer Science*, 2015.

[17] L. I. Ning, Z. A. Hao, and L. I. Yan, "Implementation and simulation research on openflow network architecture [J]," *Computer & Network*, 2014.

[18] Hping3, http://www.hping.org/hping3.html.

[19] RStudio, https://www.rstudio.com.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 9649643, 15 pages
https://doi.org/10.1155/2018/9649643

Research Article

OverWatch: A Cross-Plane DDoS Attack Defense Framework with Collaborative Intelligence in SDN

Biao Han, Xiangrui Yang, Zhigang Sun, Jinfeng Huang, and Jinshu Su

College of Computer, National University of Defense Technology, Changsha, China

Correspondence should be addressed to Xiangrui Yang; yangxiangrui11@nudt.edu.cn

Received 28 September 2017; Accepted 18 December 2017; Published 24 January 2018

Academic Editor: Chengchen Hu

Distributed Denial of Service (DDoS) attacks are one of the biggest concerns for security professionals. Traditional middle-box based DDoS attack defense is lack of network-wide monitoring flexibility. With the development of software-defined networking (SDN), it becomes prevalent to exploit centralized controllers to defend against DDoS attacks. However, current solutions suffer with serious southbound communication overhead and detection delay. In this paper, we propose a cross-plane DDoS attack defense framework in SDN, called OverWatch, which exploits collaborative intelligence between data plane and control plane with high defense efficiency. Attack detection and reaction are two key procedures of the proposed framework. We develop a collaborative DDoS attack detection mechanism, which consists of a coarse-grained flow monitoring algorithm on the data plane and a fine-grained machine learning based attack classification algorithm on the control plane. We propose a novel defense strategy offloading mechanism to dynamically deploy defense applications across the controller and switches, by which rapid attack reaction and accurate botnet location can be achieved. We conduct extensive experiments on a real-world SDN network. Experimental results validate the efficiency of our proposed OverWatch framework with high detection accuracy and real-time DDoS attack reaction, as well as reduced communication overhead on SDN southbound interface.

1. Introduction

Distributed Denial of Service (DDoS) attacks in TCP/IP networks are typically explicit attempts to disrupt legitimate users access to services, which are often launched by botnet computers that are simultaneously and continuously sending a large number of service requests to the victims [1]. The victims either respond so slowly as to be unusable or crash completely. According to Arbor Networks, which offers services to protect against DDoS attacks, they observed over 124,000 DDoS attacks per week since 2016, and they believe this number is growing rapidly [2]. Besides, since breaking the 100 Gbps barrier in 2010, DDoS attacks are also increasing in size, making them more and more difficult to defend against. Therefore, protecting network-wide resources from these frequent and large volume DDoS attacks necessitates the research community to focus on developing high-efficient defense frameworks that can be appropriately deployed in time.

Former DDoS attack defense in traditional networks involves the use of middle-box devices, which are generally complicated integration of customized hardware and software [3–6]. Although they are superior in defense performance, it is found that middle-box based DDoS attack detection is inflexible with network evolution, for example, hard to support new network architectures or protocols. Moreover, these devices are usually independently deployed in a network and have different communication interfaces. This hinders them from a holistic perception of network status, which is becoming extremely critical for network-wide defense against increasingly frequent and large volume DDoS attacks [7].

Recently, extensive research efforts have been conducted to apply software-defined networking (SDN) in diagnosing and defending DDoS attacks in a global point of view [8–12]. Different from traditional networks and information-centric networks (ICN) [13], in which the forwarding and routing decision can only be made locally, the centralized controller in SDN can quickly install reaction rules on switches and run DDoS attack defense applications without additional cost of middle-box devices. In this context, DDoS attacks

can be detected and defeated in an early stage. However, existing approaches which build DDoS attack defense applications upon the control plane are challenging to provide high defense performance. On the one hand, DDoS attack detection methods often require analysis techniques that are more advanced than de facto SDN data plane allows. Thus, the controller needs to poll flow statistics or packets from data plane switches frequently for attack detection and botnet location [8, 10, 14, 15], which increases southbound overhead and detection delay significantly. On the other hand, the potential advantages in exploiting collaborative intelligence of SDN have not been well investigated as effective DDoS attack defense requires extremely accurate detection and rapid reaction in both. Otherwise, it may result in SDN controller saturation attack in the worst case, as discussed in [16, 17].

In this paper, in order to protect hosts and servers from high volume DDoS attacks inside a particular network (e.g., autonomous system), we design and implement a high-efficient cross-plane DDoS attack defense framework with collaborative intelligence in a pure SDN environment, called OverWatch. OverWatch overcomes the aforementioned problems of existing SDN-based methods by collaboratively splitting defense functionalities across data plane and control plane and enabling both planes with abilities to intelligently detect and react to DDoS attacks in cooperation. The main difference between traditional frameworks in SDN and OverWatch is that we take the data plane into consideration for cross-plane optimization. In the proposed OverWatch framework, defense procedure is divided into two phases: detection phase and reaction phase.

In the detection phase, a lightweight flow monitoring algorithm is proposed to serve the data plane as DDoS attack sensor. We focus on two key features of DDoS attack traffic: volume feature and asymmetry feature. The proposed flow monitoring algorithm captures DDoS attack traffic in a coarse-grained manner by polling the values of SDN switch counters. On the control plane, a machine learning based DDoS attack classifier and a botnet tracking algorithm are utilized to locate a DDoS attack in finer granularity, for example, attack type and botnet locations. Specifically, features extracted from attack traffic and holistic information of the network are fed into DDoS attack classifier and botnet tracker, respectively, to determine the attack type and botnet locations.

In the reaction phase, based on the results obtained from the detection phase, a novel defense strategy offloading mechanism is proposed to enable DDoS attack defense actuators to be executed on the SDN switches automatically. Thus, SDN controller can be free from conducting specific defensive actions, resulting in a dynamic attack reaction efficiency. More specifically, we concentrate on exploiting the computational resources of switch CPUs and the flexibility of southbound interface, in order to deploy defense actuators on the switches which are closest to the botnet.

The main contributions of this paper can be summarized as follows:

(i) We design a cross-plane DDoS attack defense framework in SDN that exploits collaborative intelligence between data plane and control plane with high defense efficiency.

(ii) We develop a collaborative DDoS attack detection mechanism, which consists of a coarse-grained flow monitoring algorithm on the data plane and a fine-grained machine learning based attack classification algorithm on the control plane.

(iii) We propose a novel defense strategy offloading mechanism to dynamically deploy defense applications across the controller and switches, by which rapid attack reaction and accurate botnet location can be achieved.

(iv) We conduct extensive experiments on a real-world network with a FPGA-based OpenFlow switch prototype, a Ryu controller, and laptops generating DDoS attack traffic. Experimental results validate the efficiency of our proposed OverWatch framework with high detection accuracy and real-time DDoS attack defending reaction, as well as reduced communication overhead on SDN southbound interface.

The rest of this paper is organized as follows. Section 2 covers background and motivation of this paper. Section 3 presents the architecture of our proposed OverWatch framework. Sections 4 and 5 are mechanisms of two phases (detection phase and reaction phase) in OverWatch. Section 6 presents the experimental results. Finally, this paper is concluded in Section 7.

2. Background and Motivation

2.1. Middle-Boxes Based Defense Mechanisms. Characteristics of DDoS attacks have been widely studied, and researchers have proposed various methods to detect/defend DDoS attacks. Traditionally, DDoS attack defense applications are deployed on middle-box devices [3, 4, 19], which are specialized equipment or software that detects and reacts to DDoS attacks from a single spot on the network. The middle-boxes can provide high DDoS attack detection performance. However, due to various interfaces provided by them, different middle-boxes seldom share information, causing them and network operators to lack holistic view of DDoS attacks [20]. Mahimkar et al. in [19] proposed a method to deploy middle-boxes dynamically in the protected network, but without the global perspective of a network, where deploying them could be a nerve-racking problem. For example, assuming an attack is detected on multiple middle-boxes alone the attack trace, the most effective defense strategy would be processing malicious packets on the devices close to the source side. However, attack trace-back could not be accomplished without global information of network.

2.2. SDN-Based Defense Mechanisms. SDN provides a standard interface for a centralized controller to manage each switch under control remotely. This enables the SDN controller to obtain the entire network information and to make defense strategies in holistic views [8–11, 15, 18, 21]. In the field of network monitoring, many researches focus on how

to reduce monitoring overhead while ensuring accuracy [11, 14, 22–24]. Among them, Braga et al. in [8] proposes using Support Vector Machine (SVM) to detect DDoS attacks on the SDN controller. Chowdhury et al. in [14] proposes Payless, which includes an OpenFlow monitoring method based on an adaptive statistics collection algorithm. It can reduce the bandwidth of southbound channel but the accuracy is also reduced. Xu and Liu in [9] proposes a method to control granularity of flow by utilizing prefix masks to reduce the consumption of Ternary Content-Addressable Memory (TCAM) [25] resources while detecting DDoS attacks. Zhang in [11] proposes a prediction-based method to control the granularity of measurement while detecting abnormal traffic in order to reduce the monitoring overhead.

Moreover, many DDoS attack defense methods and systems are proposed based on SDN [26, 27]. Fresco [26] is a typical SDN-based security framework. It can poll statistics from data plane to detect different attacks. Once malicious behaviors are detected, it pushes the defense logic by installing forwarding rules on data plane switches. For example, if SYN flood attacks are detected by the defense application, the controller modifies switch rules to redirect suspected flow onto control plane to filter out malicious packets from normal ones.

2.3. Motivation of Cross-Plane Collaborative Defense. Although significant achievements have been made along this line, for example, Shin et al. in [16] enable SDN switches with more functionalities in detecting and defending SYN floods to eliminate the bottleneck between data plane and control plane, two critical problems of the existing SDN-based DDoS attack defense methods need to be pointed out here. First, both of detection and reaction process for DDoS attacks require to upload malicious traffic to the centralized controller, which introduces large amount of overhead for southbound interface and workload to the controller. Second, the controller-based DDoS attack defense mechanism breaks the initial idea of the separation of the control plane from data plane devices, as it requires the controller to process malicious packets directly. Such issues prevent SDN controller to be an intelligent centre while conducting DDoS attack defense.

Ideally, the controller in a well-defined DDoS attack defense framework should concentrate on attack analysis (e.g., attack classification and traffic trace-back). Thus, instead of implementing certain defense applications, the controller should be responsible for conducting fine-grained attack detection and making high level defense strategies, leveraging its global view of the whole network and abundant computational resources. Moreover, as data plane is where packets are proceeded, the switches in SDN should be enabled with new packets processing functions for DDoS attack detection and reaction, which are not implemented by current SDN-based DDoS attack defense approaches so far. Fortunately, most SDN switches (e.g., OpenFlow switches) consist of one or more CPUs running an operating system with abundant computational resource that is currently far from utilized [28, 29]. This inspires us to liberate the controller from heavy traffic and exploit the underutilized computing capabilities in switches to perform specific DDoS attack defense functions. Therefore, this goal could be achieved by modifying existing SDN devices. Sonchack et al. in [30] proposed a framework enabling security mechanisms to be loaded from controller to switches dynamically. Motivated by their work of leveraging computational resources on SDN switch CPUs to conduct defense mechanisms, we aim to design a collaborative DDoS attack defense framework. By exploiting the intelligence of the central controller, we focus on designing fine-grained attack detection mechanisms and automatic defense reaction by analyzing the detected DDoS attack traffic.

3. Proposed OverWatch Framework

3.1. Architecture Overview. As illustrated in Figure 1, the architecture of our proposed OverWatch framework is inspired by Knowledge Plane [31]. In OverWatch, DDoS attack sensors and defense actuators run on data plane switches. Meanwhile, DDoS attack classifier (responsible for classifying attacks), a botnet tracker (responsible for locating sources of attack traffic), and the library for defense actuators are located over the control plane.

Once OverWatch starts running, DDoS attack sensors keep monitoring every flow on the data plane constantly. If any abnormal flow is captured (i.e., DDoS attack traffic), the specific switch notifies control plane with information including an alert message and occurring attack features. Over the control plane, OverWatch leverages uploaded attack features to find out DDoS attack types and its global perspective to locate the attack sources. Next, the controller requests defense actuator library to implement specific defense actuators on the switches that are close to botnet. Last but not least, the loaded actuator will be executed on the specific switch, defending against certain DDoS attacks in the first place. After the attack is eliminated, the defense actuators used for defense will be removed from certain switches.

Our ultimate objective is that the network can detect DDoS attack threats accurately and react to them automatically. To achieve this, defense ability needs to be introduced both into control plane and data plane. We introduce design of the data plane and control plane in the following subsections, respectively.

3.2. Data Plane Design. The data plane in OverWatch is not just a group of forwarding entities, inside which there is a group of software sensors and actuators to detect and react to DDoS attacks. This leads to our three key functionalities to enhance the SDN switches: First, the data plane switches should be capable of capturing main features of DDoS attacks. Second, after the reaction strategies are made by the control plane, reaction functionalities should be loaded from control plane to data plane dynamically. Finally, these actuators can be executed on data plane to filter out attack packets. We introduce these key functionalities below.

3.2.1. DDoS Attack Sensor. We propose an algorithm which runs on the data plane devices to detect DDoS attacks as a

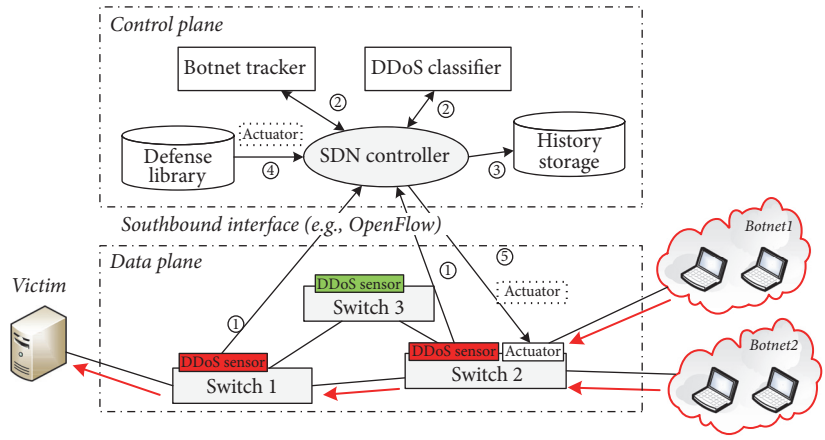


Figure 1: The architecture and workflow of OverWatch.

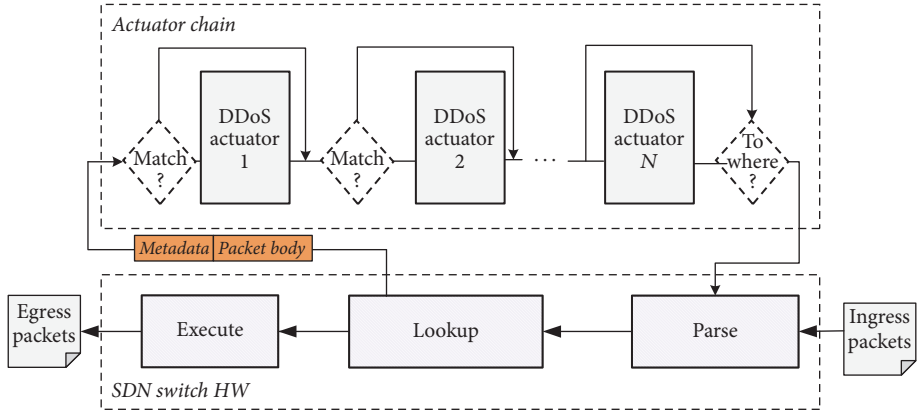


Figure 2: The process procedure of actuator chain when multiple actuators are loaded on the same switch.

DDoS attack sensor. Generally, there are plenty of approaches that are able to capture key features of DDoS attacks, for example, large volume of traffic and asymmetry in two-way traffic. However, in the context of SDN, the limitation of low-end CPU on SDN switch compared with middle-boxes causes most of them to be inappropriately deployed.

Thus, we propose a lightweight flow monitoring algorithm which recognizes DDoS attacks through reading hardware counters to server as DDoS attack sensors in OverWatch. The details of this algorithm are illustrated in Section 4. Here, we only list the two functions that can be completed by DDoS attack sensors: (1) capturing changes in flow characteristics without inspecting packets and (2) recording the DDoS attack traffic by physical port or flow ID. By this means, the sensors on the data plane are capable of capturing DDoS attacks in the first place.

3.2.2. DDoS Attack Defense Actuator. The DDoS defense actuators, to be specific, are a set of switch software tools to conduct various defense mechanisms independently. Different from traditional SDN switches, which are only capable of performing basic match-action processes, switches in OverWatch are enabled with different packet processing mechanisms by using software resources.

Various types of DDoS attacks may be conducted simultaneously to maximize the attack effect. On this occasion, multiple actuators are required to run on a single switch. Thus, we design the actuator chain, which aims to enable multiactuators to work independently, as shown in Figure 2. When a SDN switch startup, the agent process initializes an empty linked list. Once an recently loaded actuator is compiled, it is added to the tail of the list. When multiple actuators link into the list, a chain of actuators forms. Packets from hardware firstly enter the head of the chain. If the metadata contains a packet that matches the specific ID of the current actuator, this actuator will process the packet and send it back to hardware with a modified metadata (revealing the specific hardware module sent to). Otherwise, the packet will bypass the current actuator until a matched actuator ID is found. By this means, defense actuators in the same switch are able to work independently.

3.2.3. Defense Strategy Enabling. Once attacks are identified, the control plane makes a set of strategies to react. Enabling defense actuators on the data plane dynamically is a key step for executing these strategies.

The enabling procedure is inspired by the work of OFX but there are some key differences between them. First, there

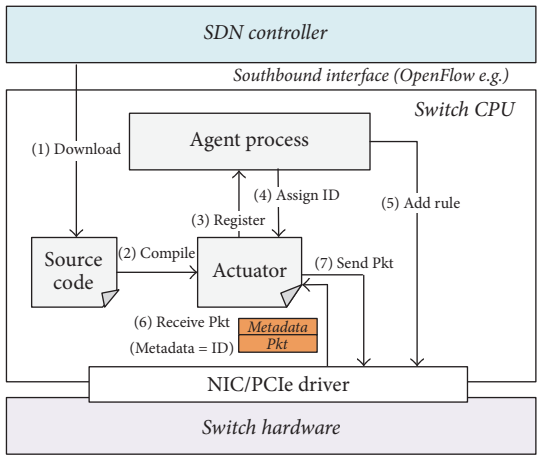


Figure 3: The procedure of defense enabling in typical SDN switches.

is no flow table maintained on the software so not all the packets need to be redirected to the software. Second, the packets redirected to the software will be processed by a specific actuator according to a metadata. The defense strategy enabling mechanism can be illustrated as Figure 3. Firstly, source code of a defense actuator is loaded from controller to local memory of a designated switch. Then, the code is compiled in the embedded operating system (usually Linux-based). Afterwards, the actuator registers to the agent process, during which an exclusive ID is assigned to the actuator. Furthermore, before the startup function of the actuator is executed, it sends a standard message to the agent process, indicating the specific packet types it processes. The agent process adds a high priority rule to the hardware match table, accordingly. In this way, packets that match such rule are polled from hardware with metadata navigating to the specific actuator. Once the above steps are completed, the actuator is executed to perform specific DDoS attack defense function.

3.3. Control Plane Design. The control plane is brain to OverWatch. It inspects the current DDoS attack (e.g., attack types and its traces) and makes proper strategy to defend it. According to our overall objective, the heart of the control plane is its intelligence ability to inspect current DDoS attack and reason proper strategies, which means the control plane should be able to (1) classify DDoS attacks and (2) track the botnets. To be specific, on the one hand, the control plane should determine exact attack types (SYN flood, UDP flood, DNS flood, etc.). On the other hand, it should also locate sources of occurring attack so as to defend DDoS attacks from the source, which proves to be more effective than defending from the destination. This argues that the control plane is responsible for the following three functionalities.

3.3.1. Attack Classification. As we use different defense actuators according to particular attack types, in order to perform defense mechanism more effectively, OverWatch is required to identify attack types firstly.

To achieve this, when DDoS attack traffic is firstly captured by data plane sensors, the abnormal traffic matching a specific rule is mirrored (by sampling) for traffic feature extraction. In order to reduce overhead of southbound interface in OverWatch to a greater extent, feature extraction is conducted on the switch software, rather than on the controller. Then the features are polled to the controller for classification. On the controller, there runs a DDoS attack classification module that leverages the extracted traffic features as input to verify the attack type. To guarantee the accuracy and reduce the false-positive rate during classification, a machine learning method is utilized in this module, which we will demonstrate in Section 4.

3.3.2. Botnet Tracking. Botnet tracking is another key issue in DDoS attack defense as it determines where the defense actuators should be deployed. Generally, DDoS attack is conducted by several botnets. As attack flows travel closer to the victim, the malicious traffic becomes larger due to traffic merger. This prevents us from effective defense measurements. In order to process attack traffic effectively, actuators should be deployed at positions close to botnets. This argues that the controller in OverWatch is ought to locate switches that are close to botnets.

Fortunately, this can be accomplished by leveraging holistic info of the network topology maintained by the controller and data from malicious packets received from data plane switches. In the next section, we propose a flexible botnet tracking algorithm suitable to be deployed on the SDN controller, which is able to locate the group of switches that are in the upstream of attack traffic.

3.3.3. Attack Reaction. When attack types and botnet locations of a DDoS attack are both determined, the control plane needs to perform highly automatic defense reaction immediately. In OverWatch, the control plane reacts by loading specific defense actuators onto directed data plane devices.

As shown in Figure 4, the reaction procedure for attacks in a typical SDN controller is quite straightforward. The defense library module, which contains various source codes of DDoS attack defense actuators, firstly registers to event listener. Once an attack is determined, a 2-tuple {`DPID, AttackType`} (DPID [32] is used to represent Data Path IDentity in the context of SDN) is sent to the event manager, indicating the defense strategy made by the built-in applications. This tuple is then received by the defense library. The defense library loads source code of specific actuator which matches AttackType in the received 2-tuple and notifies defense enabler. Finally, the actuator is loaded to designated switches through southbound channel. In order to support such mechanisms, certain modifications need to be done for existing SDN controllers, which we will describe in Section 5.

4. Detection Phase of OverWatch

As OpenFlow [32] is the leading reference implementation of the SDN paradigm, it is reasonable to implement OverWatch

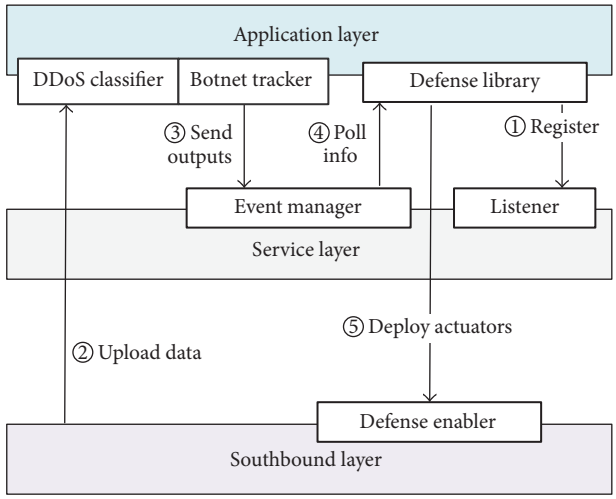


Figure 4: Workflow of attack reaction in a general SDN controller (i.e., Ryu controller). Be noted that this is only a sketch map for three-layer structured SDN controller.

in such an environment. Therefore, in this section, we introduce how we implement OverWatch into a typical SDN controller (Ryu controller [33]) and FPGA-based OpenFlow switches [18]. To describe the workflow of our OverWatch prototype clearly, we divide the working process of OverWatch into two phases: detection phase and reaction phase. The main goal in detection phase is to classify attack types as well as locate the botnets. We describe this phase in detail as follows.

4.1. Cross-Plane Attack Detection. The workflow in detection phase is shown in Figure 5. Firstly, the DDoS attack sensor constantly monitors data plane traffic by reading countervalues of each flow periodically. If attack flows are captured, the sensor notifies the OpenFlow switch agent of the specific flow ID to indicate the abnormal flow. Then, the switch agent modifies the action of hardware lookup table by a `OFPT_FLOW_MOD` message (defined in OpenFlow specification since 1.0) to mirror the sample packets from the abnormal flows onto local memory. The buffered packets have two uses: First, they are copied by the software-defined feature extraction module (SDFE), which extracts key features of different packets for attack classification on the control plane. Second, the packets themselves are also obtained by switch agent and sent to the controller for botnet tracking.

After the DDoS attack data (i.e., abnormal flow ID, sampled packets, and traffic features) is sent to the control plane encapsulated in a `OFPT_PKT_IN` message (also defined in OpenFlow specification since 1.0), it is firstly received by the OpenFlow control agent. Then, this agent extracts packet payload and passes it to the event manager. (*NB*. In Ryu, event manager is responsible for distributing messages received from data plane.) Afterward, the data is split into two parts, which are data related to traffic features and data related to botnet tracking (i.e., a {`DPID, FLOW_ID, Pkt_Buff`} three-tuple). The above two kinds of data are polled by DDoS attack classifier and botnet tracker, respectively, inside which the DDoS attack type and first-hop-switch of current DDoS attack are both determined.

From aforementioned, the detection phase is divided into two stages: a coarse-grained data plane detection stage and a fine-grained control plane detection stage. We discuss approaches we applied in both stages below.

4.2. Coarse-Grained Detection on Data Plane. As aforementioned, the data plane is where packets are forwarded; leveraging computing resources on the data plane to determine an attack coarsely and locally is quite reasonable. Therefore, on the data plane, we first present a lightweight flow monitoring algorithm that we utilize as a DDoS attack sensor on switches. It runs on the switch software as a monitor thread. Unlike many other monitoring methods, this algorithm aims to extract the key features of DDoS attack traffic by means of polling countervalues from an OpenFlow switch.

Generally, there are fundamental differences between a typical DDoS attack and normal network behaviors that we leverage to monitor DDoS attacks. Large traffic rate is one important feature for DDoS attacks. Moreover, during an attack, there is also huge rate difference between flows coming into a victim server and flows out of the server. They are defined as volume feature and asymmetry feature. Numerous researches have drawn the fact that typical DDoS attacks could be determined by verifying the above two features from traffic.

Fortunately, the above two features can be determined by polling switch countervalues from hardware pipeline. We define $C_{t_n}^{\text{Byte}}$ and $C_{t_n}^{\text{Pkt}}$ as the byte and packet count of a specific flow at time t_n. The two features can be expressed as follows.

Byte Count per Second (B_{t_n}) *at Time* t_n. It describes the average byte rate of a flow, port, or switch between time t_{n-1} and t_n:

$$B_{t_n} = \frac{C_{t_n}^{\text{Byte}} - C_{t_{n-1}}^{\text{Byte}}}{t_n - t_{n-1}}. \tag{1}$$

Packet Count per Second (P_{t_n}) *at Time* t_n. It describes the average packet rate of a flow, port, or switch between time t_{n-1} and t_n:

$$P_{t_n} = \frac{C_{t_n}^{\text{Pkt}} - C_{t_{n-1}}^{\text{Pkt}}}{t_n - t_{n-1}}. \tag{2}$$

Byte Count Asymmetry ($A_{t_n}^{\text{Byte}}$) *at Time* t_n. It describes the average byte rate asymmetry of pair-flow or port between time t_{n-1} and t_n:

$$A_{t_n}^{\text{Byte}} = \frac{B_{t_n}^{\text{in}}}{B_{t_n}^{\text{out}}}. \tag{3}$$

Packet Count Asymmetry ($A_{t_n}^{\text{Pkt}}$) *at Time* t_n. It describes the average packet rate asymmetry of pair-flow, port, or switch between time t_{n-1} and t_n:

$$A_{t_n}^{\text{Pkt}} = \frac{P_{t_n}^{\text{in}}}{P_{t_n}^{\text{out}}}. \tag{4}$$

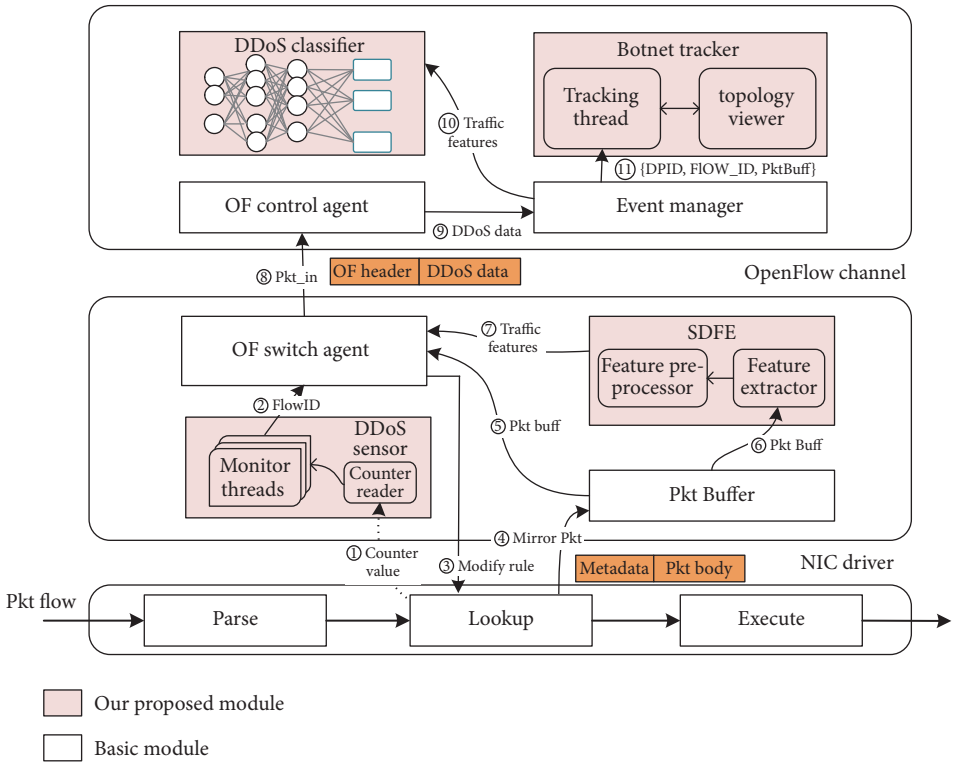


Figure 5: Implementation and workflow of OverWatch in detection phase.

We present our prediction-based algorithm to capture great changes of the above four metrics. This algorithm leverages previous metric samples from a specific flow to estimate a future value range. If the actual values of the four metrics fall into the range we predict, this indicates that the current flow is normal. Otherwise, the deviation between the predicted values and observed values indicates an abnormal flow caused by a DDoS attack.

Specifically, we leverage WMA (Weighted Moving-Average) to calculate prediction value for each metric. And Pauta criterion in Gaussian distribution is also utilized to get a reasonable prediction range. The pseudocode of this algorithm is shown in Algorithm 1.

4.3. Fine-Grained Detection on Control Plane. As a centralized and often high-performance platform, the control plane holds advantages of abundant computing resources and holistic info of the whole network. Thus, on the control plane, two functionalities are developed: DDoS attack classification and botnet tracking. Both of them are essential for attack reaction as they determine which actuator is to be deployed and on which data plane switch it is to be deployed. We introduce our machine learning based classification model as well as a lightweight botnet tracking algorithm below.

4.3.1. Autoencoder-Based Attack Classification. Machine learning has gained much attention in the community of network security as it improves the accuracy and reduces

globals: $\mathbf{V}[n]$ //Vector List of DDoS attack feature metrics
t_n //Current time
n //Number of Vectors in the list
while 1 **do**
if $t_n = t_{n-1} + \Delta t$ **then**
Update the list $\mathbf{V}[n]$ of history records.
for all $i \in \{1, 2, 3, 4\}$ **do**
Use WMA to calculate the prediction value $\mathbf{V}_{n+1}^{\text{predict}}$ for next time interval:

$$\mathbf{V}_{n+1}^{\text{predict}} = \sum_{i=1}^{n} \lambda_i \mathbf{V}_i^{\text{actual}} \quad \sum_{i=1}^{n} \lambda_i = 1$$

Use ratio metric R^{predict} to compare prediction value and actual value.
Calculate ideal value v_i^{ideal} and standard deviation σ for ratio metric.
Use Pauta criterion to calculate the prediction range:

$$R_u \longleftarrow v_i^{\text{ideal}} + 3 \times \sigma$$
$$R_l \longleftarrow v_i^{\text{ideal}} - 3 \times \sigma$$

end for
if each R^{predict} is out of the (R_l, R_u) range **then**
Trigger alert to controller;
else
Continue;
end if
end if
end while

Algorithm 1: Lightweight flow monitoring algorithm.

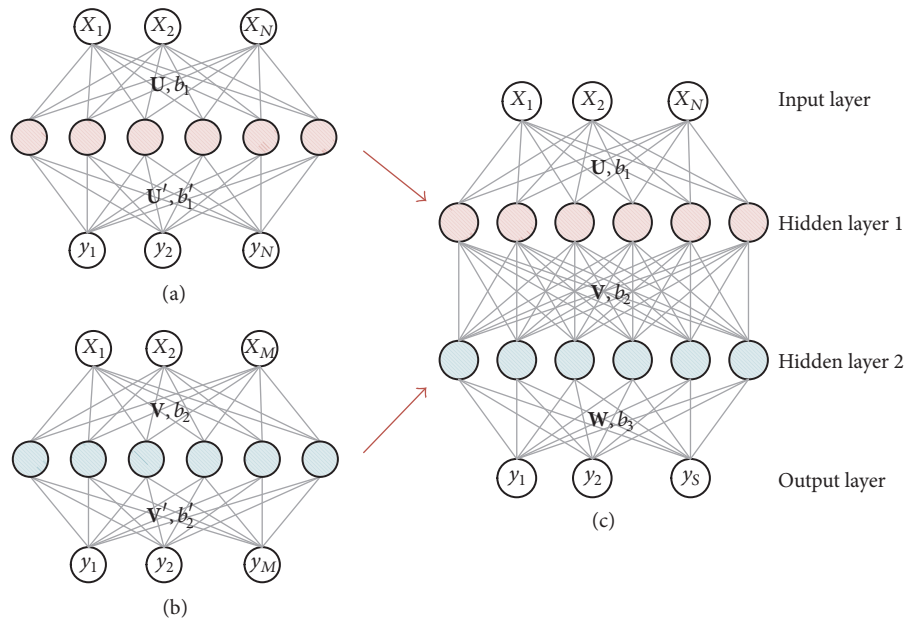


Figure 6: Schematic representation of autoencoder and DDoS attack classifier model: (a) autoencoder for A_1; (b) autoencoder for A_2; (c) the model of DDoS attack classifier.

false-positive rate while classifying different types of abnormal traffic. To determine the attack type from real-time extracted traffic features, a machine learning method, combined with autoencoder [34] and softmax classifier [35], is utilized in the module of DDoS attack classifier.

As shown in Figure 6, each autoencoder contains three layers: input layer, hidden layer, and output layer. Two autoencoders (A_1 and A_2) are stacked with each other in a way that the outputs of first hidden layer are fed into the inputs of the second. Then, the outputs of the second hidden layer are fed into a softmax classifier. Finally, all layers stacked together, forming the DDoS attack classifier. Generally, if the extracted traffic features are fed into the input layer, the output vector of the model indicates to which attack type the features belong. We introduce the structure below.

The autoencoder has three layers: an input layer of N nodes for a record of N features (i.e., $X = \{x_1, x_2, \ldots, x_N\}$), a hidden layer of M nodes for learning key patterns of input record, and a output layer of N nodes for reconstruction of the input (i.e., $\widehat{X} = X$). The network finds optimal values of weight matrix (i.e., $U \in R^{N\times M}$ and $U' \in R^{M\times N}$) and bias vector (i.e., $b_1 \in R^{N\times 1}$ and $b_1' \in R^{M\times 1}$) together, while trying to learn the key patterns of an input record (e.g., a DDoS attack record). The second autoencoder feeds the outputs of the first one as its input. It uses the same methods to calculate the optimal weight matrix $V \in R^{P\times N}$ and bias vector $b_2 \in R^{P\times 1}$.

Then, a softmax classifier builds a mapping relationship between the hidden layer of the former autoencoder (i.e., $H = \{h_1, h_2, \ldots, h_M\}$) and S types of DDoS attacks (i.e., $Y = \{y_1, y_2, \ldots, y_S\}$). Similarly, This network finds optimal values of weight matrix (i.e., $W \in R^{M\times S}$) and bias vector (i.e., $b_3 \in R^{S\times 1}$) too, while polling the output close to the label value, which indicates the real DDoS type. Before this model is able to classify any record collected from DDoS traffic, each layer needs to be trained with backpropagation algorithm [36], separately. Then, they are stacked together and fine-tuned to improve the performance of the entire model. The brief training process for the first autoencoder is shown in Algorithm 2, and the other two layers share similar training process.

After the training process with historical DDoS attack dataset, this model can be utilized to perform attack classification with run traffic records.

4.3.2. Collaborative Botnet Tracking. The collaborative botnet tracking aims to locate the switches close to the botnets, thus making the defense actuators more effective in defending DDoS attacks. We propose a botnet tracking algorithm based on the collaboration of the data and control plane and implement it in the controller as a built-in module, called botnet tracker.

Before diving into the details of the algorithm, two prerequisites need to be stressed out: The first one is that the whole network info (link state, topology info, forwarding rule, etc.) is maintained on the controller. The second prerequisite is that the info maintained by the controller can be accessed by other built-in applications on the controller. Fortunately, both prerequisites can be satisfied by leveraging an enhanced topology viewer [37], a built-in module in Ryu. More specifically, the key procedure in the algorithm is to obtain the last hop of a sampled packet. This could be achieved by extracting the source MAC address of the packet and leveraging the network info on the controller to locate the last hop switch.

Based on above, we propose our botnet tracking algorithm. Specifically, we consider a set A consisting a set of switches that have captured DDoS attacks on themselves $A = \{a_1, a_2, \ldots, a_m\}$. And S is the total set of data plane switches a

Require: Weight matrix of the hidden layer: U
Bias vector of the hidden layer: b_1
Ensure: Training dataset e
for all number of training iterations **do**
Train the single layer autoencoder using back-propagation:
$\{X^1, X^2, \ldots, X^m\} \leftarrow$ Sample minibatch of e //get batches of traffic records
Update U, U', b, b' by using gradient descent method to minimize the loss function:

$$\text{Loss} = \left(\frac{1}{2m}\sum_{i=1}^{m}\|X^i - \widehat{X}^i\|^2\right) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^l)^2$$

end for

Algorithm 2: DDoS attack classifier training process.

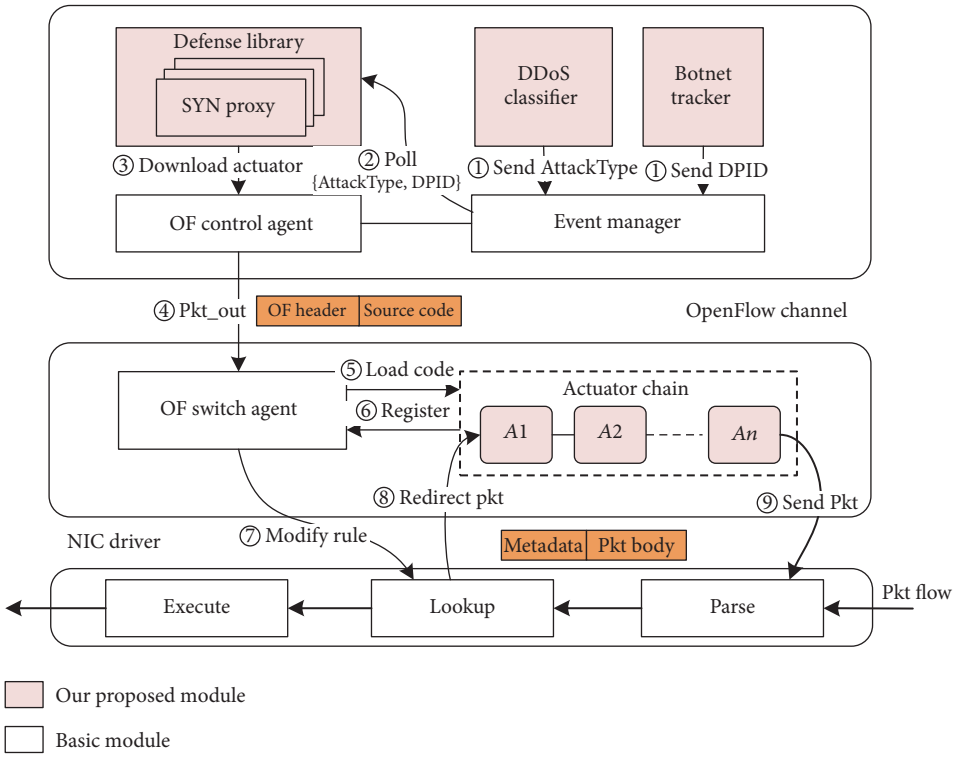


Figure 7: Implementation and workflow of OverWatch in response phase.

controller maintains $S = \{s_1, s_2, \ldots, s_n\}$. We take one element a_i from set A at a time, and use the sampled packets collected from a_i to determine the last hop switch s_k. If $s_k \in A$, then we eliminate a_i from set A. Otherwise, a_i is one of the switches we are searching for. We use this method to traverse set A and obtain a subset B ($B \subseteq A$) consisting of all a_i whose last hop is not included in A. In this way, we are able to locate all the switches which are likely close to botnets.

5. Reaction Phase of OverWatch

In this section, we express how we design and implement the reaction phase of OverWatch. After the attack type and switches that are close to botnets are addressed in the former phase, OverWatch is supposed to react to the occurring attack efficiently. Thus, first, we illustrate the workflow of the reaction phase in our prototype. Then, two reaction applications are introduced in the second part.

5.1. Attack Mitigation. The workflow of reaction phase in OverWatch is depicted in Figure 7. From aforementioned, the specific attack type and the most close-in switches can be determined in the detection phase, respectively. Then, both results are sent back to the event manager. This triggers the defense library, which has registered to the event manager, to poll up the messages. This module firstly leverages the attack

type to match a particular actuator so as to indicate the source code which is required to be loaded onto data plane. Then, it invokes functions provided by OpenFlow control agent to download the specific source code onto the switch whose DPID matches the received message from the event manager.

In OverWatch, the source code of designated actuator is encapsulated inside a `OFPT_EXPERIMENTER` message (defined in OpenFlow specification since 1.1) and sent to the specific switch through OpenFlow channel. The corresponding switch agent running on that switch receives the message and loads the codes of the actuator in the running space. Then, the source code is compiled to an executable file and then registered back to the switch agent, which later allocates an exclusive ID to the specific actuator so that it could be added to the actuator chain. Meanwhile, the actuator also notifies the switch agent of the packet type it processes. Once this is received by the agent, the agent generates a high priority flow rule and adds it to the flow table using a `OFPT_FLOW_MOD` message.

After the rule modification takes effect, packets that match the higher priority rule are redirected to the actuator chain with metadata that could match a certain ID of the actuator, in which they are going to be processed. In addition, packets sent back to the hardware are also allocated with metadata to match the lower priority rule so that they can be forwarded by the switch's original rules.

5.2. Intelligent Reaction Applications. We develop two sample applications of DDoS defense actuators to exemplify the feasibility of OverWatch. This includes SYN proxy and DNS reflection filter. These two actuators are motivated by (1) the functionality that Avant-Guard [16] proposed as a data plane extension to defend SYN flood and (2) the example DNS filter proposed in SDPA [38] to filter out DNS refection attack packets on OpenFlow switches.

5.2.1. SYN Proxy. In a SYN flood, attackers send numerous SYN packets to exhaust memory resources of victim by enforcing it maintaining a large number of semiconnected states, so the victim will not respond to affirmed connection request. To filter out these malicious SYN requests, an OpenFlow rule is preloaded to lookup table so that all SYN packets will be polled onto an actuator called SYN proxy. If a SYN packet is received by it, firstly, to prevent multiple SYN requests sending to the victim, it calculates a cookie to record the request using harsh table, and then the actuator generates a response packet sent back to the source and drops the initial SYN request packet. If, during a certain time interval, an affirmed ACK packet is received from a source that has been recorded in the harsh table, the proxy will generate TCP handshake packets to build up validation between the legal source and its destination. Otherwise, the proxy simply drops the malicious packets. In this way, the proxy eliminates the threat of SYN flood.

5.2.2. DNS Filter. Botnets in a DNS reflection attack send DNS requests to name servers using the victim host's IP address. Thus, the victim will be flooded by these massive DNS responses. To filter out these unsolicited DNS response,

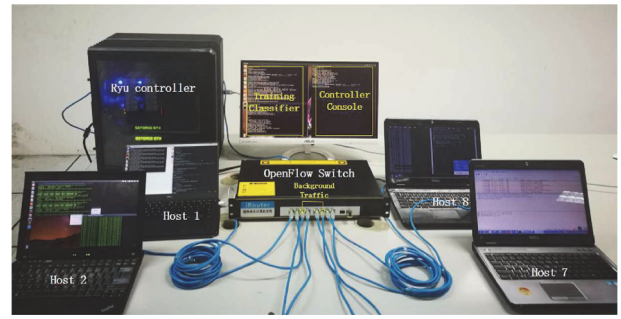


Figure 8: Diagram of our testbed network.

once the actuator (DNS filter) is loaded on the data plane, two high priority OpenFlow lookup rules, which redirect the packets whose UDP source or destination port equals 53, are loaded as well. After the redirected DNS packets are received by the filter, each DNS request packet is recorded by a five-tuple (source IP, destination IP, source port, destination port, protocol) in memory. If the upcoming packet is a DNS response packet and matches one of the tuples of request, it is sent to hardware pipeline to match a lower priority rule for forwarding. On the contrary, the filter drops the packet to protect the victim.

6. Experiment and Evaluation

6.1. Experiment Setup. To evaluate the performance of our proposed OverWatch framework, we modified a FPGA-based (Altera EP4SGX180) OpenFlow switch [28] to support the aforementioned data plane functions. We also modified Ryu controller to enable proposed controller-based mechanisms (including the adding of three built-in applications: DDoS attack classifier, botnet tracker, and defense library).

Figure 8 illustrates the testbed of our experiment. It consists of a FPGA-based OpenFlow switch prototype with 8 Gigabit ports, a 1.99 GHz Intel Celeron J1900 CPU and a 2 GB memory that runs Ubuntu 14.04 on it, a control platform with a quad-core Intel i7 CPU and a dual NVIDIA-GTX1080 GPU (used for training machine learning based classifier) with 16 GB of RAM running Ryu controller, and up to eight laptop hosts, which represent DDoS attackers, victims, and normal traffic generators, respectively.

6.2. Efficiency of Coarse-Grained Detection. In order to evaluate the performance of the algorithm running as DDoS attack sensors, we firstly add two rules to enable traffic forwarding from H_1 and H_2 to H_8 ($H_1, H_2 \rightarrow H_8$). Then, Stacheldraht [39] is utilized to conduct DDoS attacks from H_1 and H_2 to H_8 within 20 seconds. The detection results in the DDoS attack sensor, together with volume and asymmetry features of attack flow ($H_1 \rightarrow H_8$), are depicted in Figure 9. These depicted results demonstrate that the four metrics in the DDoS attack sensor are able to capture great changes in volume and asymmetry features as soon as the attack occurs, which also evidently shows the effectiveness of the algorithm.

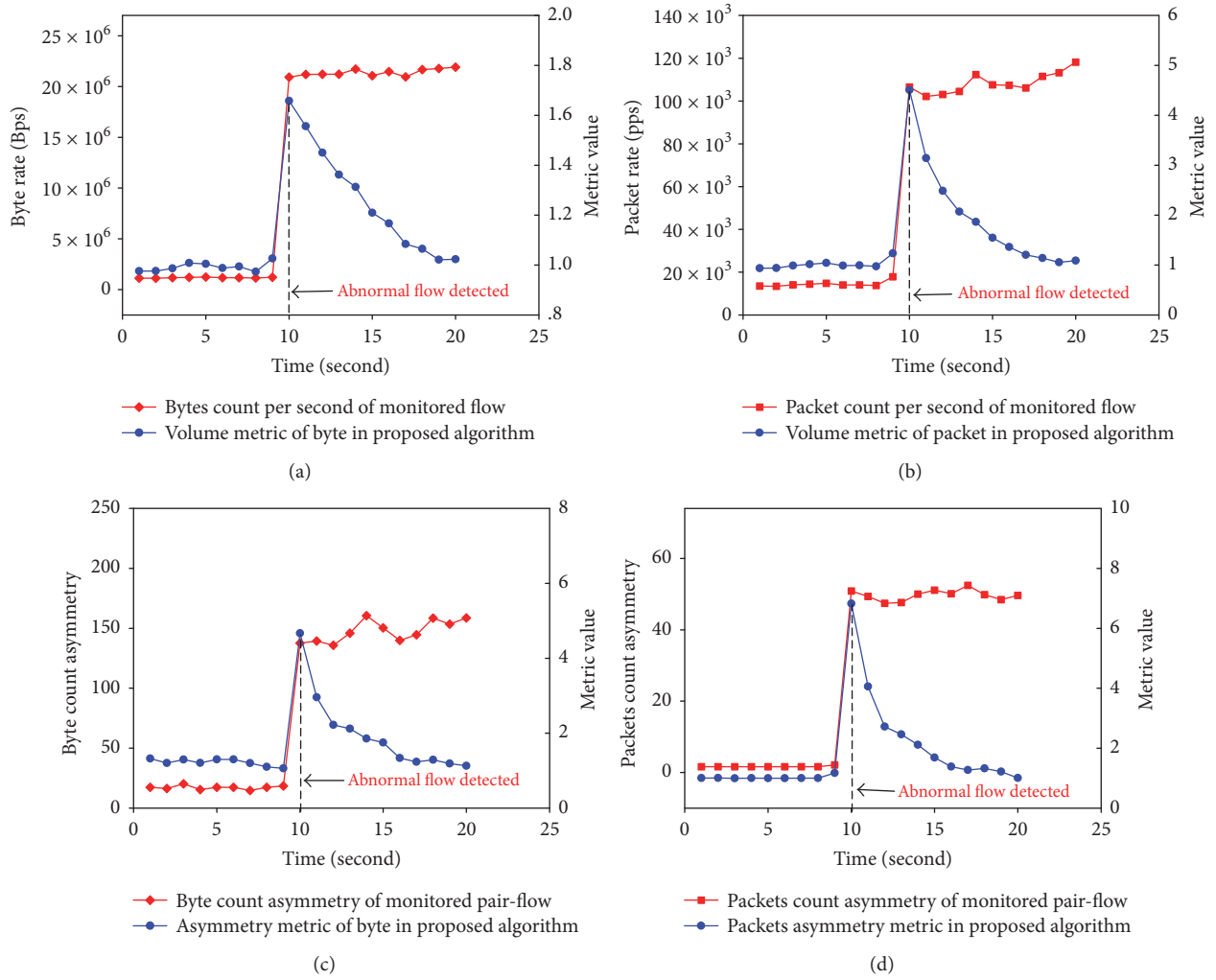


FIGURE 9: The coarse-grained detection results during a SYN flood attack [18]: (a) byte rate and volume metric of byte in monitored flow; (b) packet rate and volume metric of packet in monitored flow; (c) byte count asymmetry and asymmetry metric of byte in monitored pair-flow; (d) packet count asymmetry and asymmetry metric of packet in monitored pair-flow.

Next, we use FTP to transfer a 4 GB data block from H_1 to H_8. The detection results of four metrics are shown in Figure 10. It is found that even though three of the metrics dramatically change in the algorithm result, the asymmetry feature of packet count barely changes. This is because during the data transfer, the receiver H_8 keeps sending ACK packets to H_1, which ensures a rough equivalence of packet count in both directions. This explanation can be testified by using Wireshark to capture packets from H_1 or H_8. Therefore, such mechanisms in the proposed algorithm enable the DDoS attack sensor to reduce the misjudgment rate of DDoS attack detection.

To demonstrate the advantage of communication overhead reduction in OverWatch, we move the DDoS attack sensor to the controller side and use the same algorithm to poll switch countervalues through OpenFlow channel. We set this typical controller-based DDoS detection method as a baseline method, Besides, we also implement another mechanism, which optimizes the baseline method by utilizing an adaptive polling algorithm proposed in Payless [14] to reduce communication overhead of southbound interface. Then, we implement these three methods in a scenario where different times of DDoS attacks are conducted from random hosts within 1 minute. Wireshark is utilized to capture all the packets of southbound interface during the experiment. The total amount of southbound overhead is shown in Figure 11. It is found that OverWatch has orders of magnitude less overhead than the other two methods. This is achieved by offloading the DDoS attack sensor onto data plane. And compared with Avant-Guard, which can reach similar results, there is no modification introduced in the switch hardware.

6.3. Efficiency of Fine-Grained Detection. To demonstrate the performance of the DDoS attack classifier in OverWatch, we collected network traffic from the testbed as training dataset. Specifically, we use H_1 and H_2 to send DDoS attack traffic to H_8. Hosts from H_3 to H_6 communicated with each other randomly for web browsing, video transferring, and

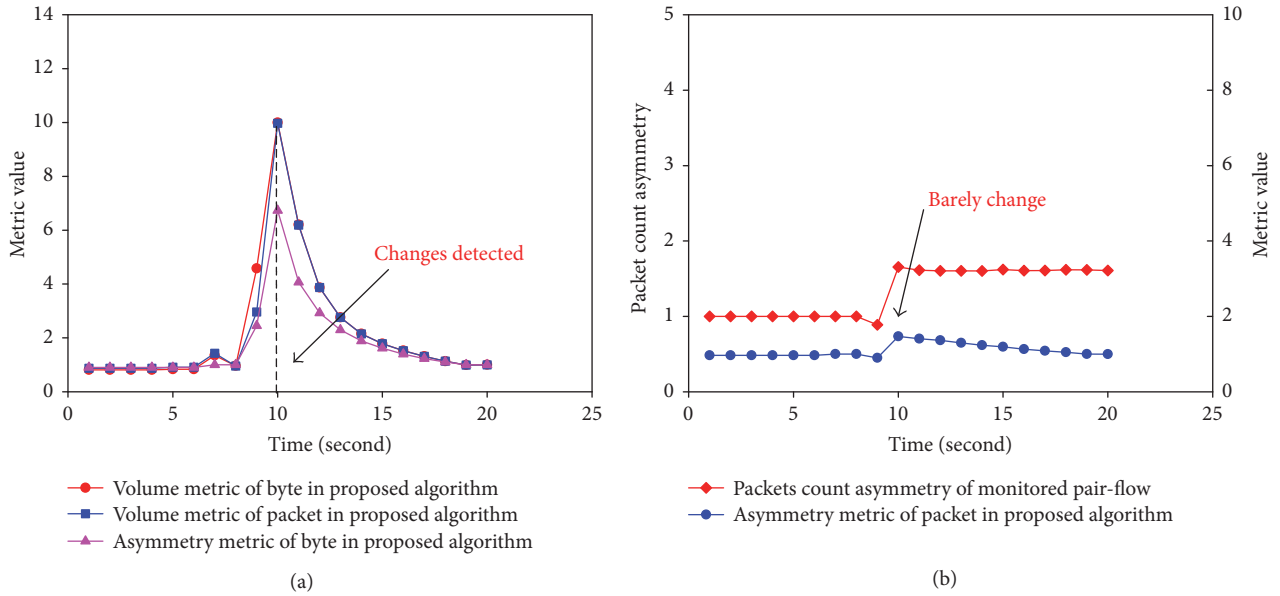


Figure 10: The coarse-grained detection results during big data block transferring using FTP [18]: (a) the changes of three metric values except asymmetry metric of packet; (b) the changes of packet count asymmetry and asymmetry metric of packet in monitored pair-flow.

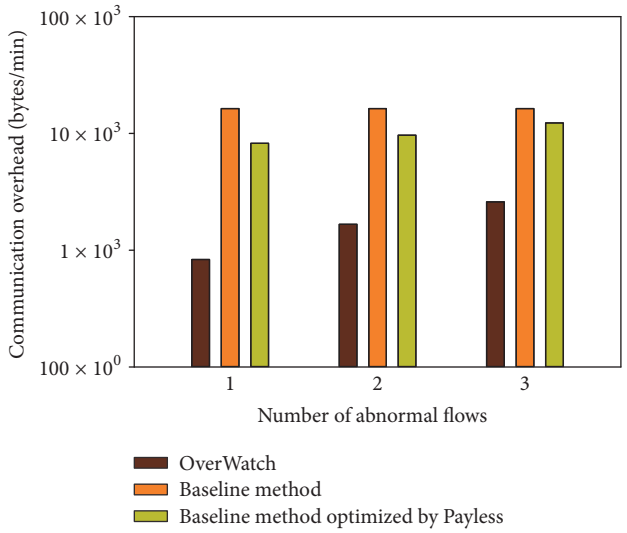


Figure 11: Overhead of southbound interface in OverWatch, baseline method, and baseline method optimized by Payless within 1 minute [18].

Table 1: Number of records in training dataset.

Traffic class	Records number	
	Training	Test
Normal traffic	17539	9824
Attack traffic		
SYN flood	2831	1566
UDP flood	2706	1591
ICMP flood	2519	1630
SYN & UDP flood	3054	1321
UDP & ICMP flood	2936	1729
SYN & ICMP flood	3144	1538

online gaming, which led to background traffic variation. We mirrored all the traffic to H_7 by tcpdump, so that the traffic can be leveraged as dataset. We collected 12 hours of traffic data in total, including 6 hours of normal traffic and 6 hours of attack traffic. DDoS attacks in the conducted experiment consist of 6 types: UDP flood, SYN flood, ICMP flood, and their permutations. They are all generated by Stacheldraht. And Table 1 shows the distribution of records in the dataset. The features we extracted from traffic flows during the classification are listed in Table 2, while Table 3 lists the parameters in our experiment. In the training and evaluation process, the features are real-valued positive numbers by digitization and max-min normalization to improve accuracy.

We evaluate the performance of the classifier using parameters including confusion matrix, precision, recall, and f-measure. In a confusion matrix, each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class. In binary classification, precision is the fraction of relevant instances among the retrieved instances, while recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. Both of them indicate the performance of the classifier. F-measure considers both parameters above to compute a score, which is the harmonic average of the parameters, where f-measure reaches its best value at 1 and worst at 0. Figure 12 illustrates the confusion matrix of our evaluation. It is observed that our DDoS attack classifier has fairly good accuracy for detecting single type of DDoS attacks, reaching about 96%. The detection accuracy for mixed attacks is lower but still reaches around 83%.

Table 2: Features extracted from different packets.

Packet type	#	Feature description
TCP	1	Fraction of TCP packets with SYN flag set
	2	Fraction of TCP packets with ACK flag set
	3	Entropy of src IP addresses
	4	Entropy of dst IP addresses
	5	Entropy of src ports
	6	Entropy of dst ports
	7	Entropy of TCP sequences
UDP	8	Fraction of dst port ≤ 1024 UDP packets
	9	Fraction of dst port > 1024 UDP packets
	10	Entropy of src IP addresses
	11	Entropy of dst IP addresses
	12	Entropy of length for UDP packets
ICMP	13	Entropy of src IP addresses
	14	Entropy of dst IP addresses
	15	Entropy of TTL values
	16	Fraction of ICMP packets in total

Table 3: Autoencoder training parameters.

Parameter	Value
Learning rate	0.1
Batch size	5
Epoch limit	3500

More specifically, we demonstrate the precision, recall and f-measure for 8 types of traffic in Figure 13. Except the mixed traffic of SYN and UDP flood as well as SYN and ICMP flood, the 3 parameters for classification of all types traffic reach above 90%, which is quite acceptable in classifying DDoS attacks in real network.

We claim that the performance of the autoencoder-based classifier is not necessarily better than other machine learning approaches, but these results demonstrate the great feasibility of leveraging machine learning approaches to serve OverWatch as a DDoS attack classifier, which is the main purpose of the evaluation above.

6.4. Performance of Attack Reaction. We also evaluated the performance of defense actuators on the data plane. We set a scenario where we redirected packets from a certain port to a defense actuator, which performs no operations to the packets, and then send the packets back to a designated port. We compare this forwarding path with another two baseline methods. The first one is direct hardware forwarding; the second one is redirecting the packets to the Ryu controller and then sending them back to a designated port. The main goal here is to evaluate how much performance gain is introduced by the defense actuator in OverWatch, compared with traditional SDN-based defense mechanisms. According

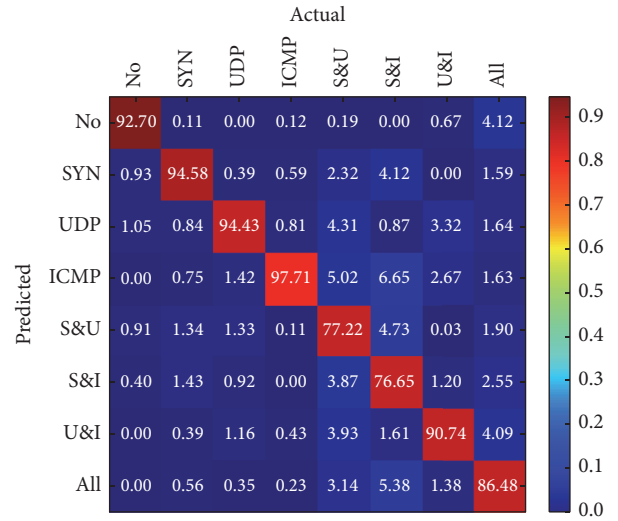


Figure 12: Confusion matrix for DDoS attack classifier in OverWatch.

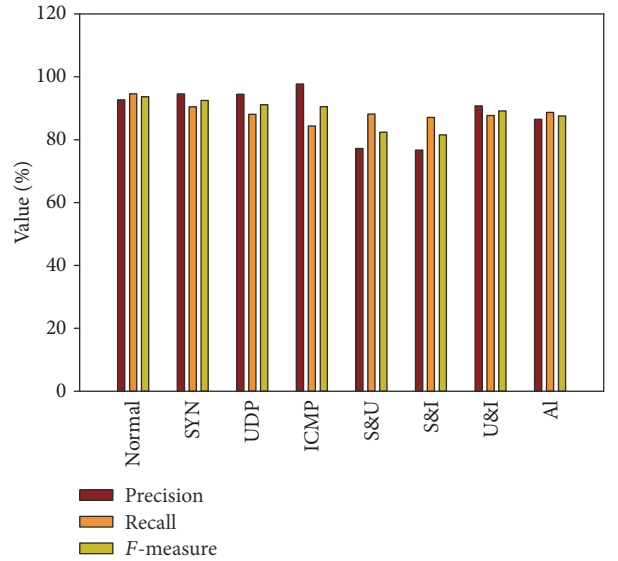


Figure 13: Precision, recall, and f-measure for the DDoS attack classifier in OverWatch.

to the evaluation result shown in Figure 14, defense actuators in OverWatch perform several orders of magnitude better than the controller-based reaction mechanisms. Moreover, though the forwarding performance for actuators is evidently lower compared with hardware path, this could be improved if high-efficient data path (e.g., DPDK [40]) is utilized for the communication between software and hardware on the switch.

7. Conclusion

To overcome the challenges of southbound bottleneck and the lack of collaborative intelligence in SDN-based DDoS attack defense mechanisms, in this paper, we introduced OverWatch, a SDN-based high-efficient cross-plane DDoS

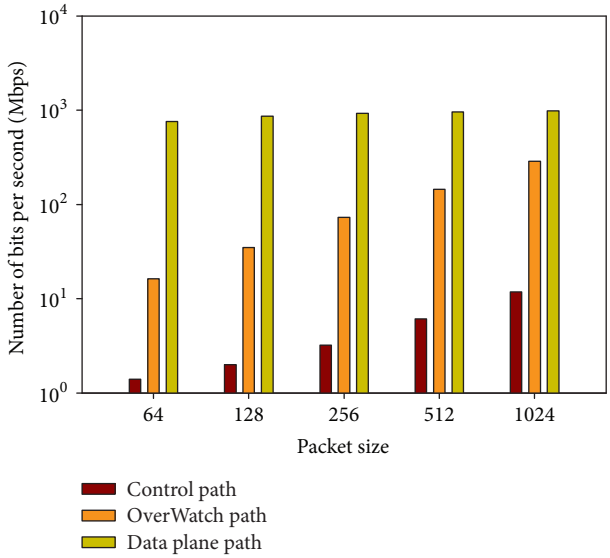


FIGURE 14: Different bits rate according to packet size using different paths in the testbed.

attack defense framework with collaborative intelligence. It is collaboratively splitting defense functionalities across data and control plane and enabling both planes to detect and defend against DDoS attacks on different levels. Through experiments, it can be concluded that OverWatch is capable of high accuracy detection and real-time defending reaction. Meanwhile, the communication overhead on SDN southbound interface is also greatly reduced. All these outcomes demonstrate the feasibility of OverWatch in large-scale networks. We anticipate that OverWatch becomes a building block in the SDN-based network security applications.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported in part by Chinese National Programs for High Technology Research and Development (863 Programs) (Grant no. 2015AA016103), the project of National Science Foundation of China (Grant no. 61601483), and Startup Research Grant of National University of Defense Technology (Grant no. JC15-06-01).

References

[1] S. Yu, Y. Tian, S. Guo, and D. O. Wu, "Can we beat DDoS attacks in clouds?" *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2245–2254, 2014.

[2] D. Bisson, "The 5 most significant ddos attacks of 2016," https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/5-significant-ddos-attacks-2016/.

[3] D. Geneiatakis, G. Portokalidis, and A. D. Keromytis, "A multilayer overlay network architecture for enhancing IP services availability against DoS," in *Proceedings of the International Conference on Information Systems Security*, vol. 7093, 2011.

[4] X. Liu, X. Yang, and Y. Lu, "To filter or to authorize: Network-layer DoS defense against multimillion-node botnets," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM'08*, pp. 195–206, August 2008.

[5] P. Mittal, D. Kim, Y.-C. Hu, and M. Caesar, *Mirage: Towards Deployable Ddos Defense for Web Applications*, 2011.

[6] H. Wang, Q. Jia, D. Fleck, W. Powell, F. Li, and A. Stavrou, "A moving target DDoS defense mechanism," *Computer Communications*, vol. 46, pp. 10–21, 2014.

[7] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDOS) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[8] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks (LCN '10)*, pp. 408–415, Denver, Colo, USA, October 2010.

[9] Y. Xu and Y. Liu, "DDoS attack detection under SDN context," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications, IEEE INFOCOM 2016*, pp. 1–9, April 2016.

[10] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proceedings of the 2015 International Conference on Computing, Networking and Communications, ICNC 2015*, pp. 77–81, February 2015.

[11] Y. Zhang, "An adaptive flow counting method for anomaly detection in SDN," in *Proceedings of the 2013 9th ACM International Conference on Emerging Networking Experiments and Technologies, CoNEXT 2013*, pp. 25–30, December 2013.

[12] Y. Cui, L. Yan, S. Li et al., "SD-Anti-DDoS: Fast and efficient DDoS defense in software-defined networks," *Journal of Network and Computer Applications*, vol. 68, pp. 65–79, 2016.

[13] S. Oueslati, J. Roberts, and N. Sbihi, "Flow-aware traffic control for a content-centric network," in *Proceedings of the IEEE Conference on Computer Communications, INFOCOM 2012*, pp. 2417–2425, March 2012.

[14] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for software defined networks," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World, NOMS 2014*, pp. 1–9, May 2014.

[15] J. Seo, C. Lee, T. Shon, K.-H. Cho, and J. Moon, "A new DDoS detection model using multiple SVMs and TRA," in *Proceedings of the EUC Workshops*, 2005.

[16] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013*, pp. 413–424, November 2013.

[17] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, "LineSwitch: Tackling Control Plane Saturation Attacks in Software-Defined Networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1206–1219, 2017.

[18] X. Yang, B. Han, Z. Sun, and J. Huang, "Sdn-based ddos attack detection with cross-plane collaboration and lightweight flow monitoring," in *Proceedings of the Global Communications Conference*, 2017.

[19] A. Mahimkar, J. Dange, V. Shmatikov, H. M. Vin, and Y. Zhang, "dfence: Transparent network-based denial of service mitigation," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[20] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "DDoS attack protection in the era of cloud computing and Software-Defined Networking," *Computer Networks*, vol. 81, pp. 308–319, 2015.

[21] K. Kalkan, G. Gur, and F. Alagoz, "Defense Mechanisms against DDoS Attacks in SDN Environment," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 175–179, 2017.

[22] M. Moshref, M. Yu, and R. Govindan, "Resource/accuracy tradeoffs in software-defined measurement," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, pp. 73–78, August 2013.

[23] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," RFC Editor RFC3176, 2001.

[24] M. Aslan and A. Matrawy, "On the impact of network state collection on the performance of SDN applications," *IEEE Communications Letters*, vol. 20, no. 1, pp. 5–8, 2016.

[25] Z. Cai, Z. Wang, K. Zheng, and J. Cao, "A Distributed TCAM coprocessor architecture for integrated longest prefix matching, policy filtering, and content filtering," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 417–427, 2013.

[26] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proceedings of the Network and Distributed System Security*, 2013.

[27] "Defense4all:tutorial," https://wiki.opendaylight.org/view/Defense4All:Tutorial.

[28] J. Mao, B. Han, Z. Sun, X. Lu, and Z. Zhang, "Efficient mismatched packet buffer management with packet order-preserving for OpenFlow networks," *Computer Networks*, vol. 110, pp. 91–103, 2016.

[29] L. Boero, M. Cello, C. Garibotto, M. Marchese, and M. Mongelli, "BeaQoS: Load balancing and deadline management of queues in an OpenFlow SDN switch," *Computer Networks*, vol. 106, pp. 161–170, 2016.

[30] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller, "Enabling practical software-defined networking security applications with ofx," in *Proceedings of the Network and Distributed System Security*, vol. 16, pp. 1–15, 2016.

[31] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the internet," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 3–10, ACM, 2003.

[32] O. S. S. Version, *Openflow Switch Specification 1.5. 1 (Protocol Version 0x06)*, 2014.

[33] F. Tomonori, "Introduction to ryu sdn framework," in *Proceedings of the Open Networking Summit*, April 2013.

[34] J. Schmidhuber, "Deep learning in neural networks: an overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[35] R. Gens and P. Domingos, "Deep symmetry networks," in *Proceedings of the 28th Annual Conference on Neural Information Processing Systems 2014, NIPS 2014*, pp. 2537–2545, December 2014.

[36] L. Wang, Y. Zeng, and T. Chen, "Back propagation neural network with adaptive differential evolution algorithm for time series forecasting," *Expert Systems with Applications*, vol. 42, no. 2, pp. 855–863, 2015.

[37] Y. Hideki, "Topology viewer," https://github.com/osrg/ryu/blob/master/doc/source/gui.rst.

[38] S. Zhu, J. Bi, C. Sun, C. Wu, and H. Hu, "SDPA: Enhancing stateful forwarding for software-defined networking," in *Proceedings of the 23rd IEEE International Conference on Network Protocols, ICNP 2015*, pp. 323–333, November 2015.

[39] D. Dittrich, *The "Stacheldraht" Distributed Denial of Service Attack Tool*, 1999.

[40] D. Intel, *Data Plane Development Kit*, 2015.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 4123736, 10 pages
https://doi.org/10.1155/2018/4123736

Research Article

Security Analysis of Dynamic SDN Architectures Based on Game Theory

Chao Qi, Jiangxing Wu, Guozhen Cheng, Jianjian Ai, and Shuo Zhao

National Digital Switching System Engineering & Technological R&D Center, Zhengzhou, Henan 450002, China

Correspondence should be addressed to Chao Qi; 13937147170@163.com

Received 26 September 2017; Accepted 26 December 2017; Published 23 January 2018

Academic Editor: Zhiping Cai

Security evaluation of SDN architectures is of critical importance to develop robust systems and address attacks. Focused on a novel-proposed dynamic SDN framework, a game-theoretic model is presented to analyze its security performance. This model can represent several kinds of players' information, simulate approximate attack scenarios, and quantitatively estimate systems' reliability. And we explore several typical game instances defined by system's capability, players' objects, and strategies. Experimental results illustrate that the system's detection capability is not a decisive element to security enhancement as introduction of dynamism and redundancy into SDN can significantly improve security gain and compensate for its detection weakness. Moreover, we observe a range of common strategic actions across environmental conditions. And analysis reveals diverse defense mechanisms adopted in dynamic systems have different effect on security improvement. Besides, the existence of equilibrium in particular situations further proves the novel structure's feasibility, flexibility, and its persistent ability against long-term attacks.

1. Introduction

SDN is a novel and promising framework which can be applied in traditional and wireless networks to achieve highly programmable switch infrastructure [1, 2]. It separates the data and control planes, which makes switches become simple data forwarding devices, and network is manipulated through logically centralized controllers [3]. It is no doubt that this processing mechanism will definitely improve the efficiency of network management and performance of network operation [4]. However, this reliance on a centralized controller can easily lead to a single point of failure if not carefully designed and implemented. Moreover, a new set of threats that are unique to SDN can render the network vulnerable especially when the control plane is compromised. Thus, in order to enhance security of SDN, different SDN architectures employing multiple controllers have been proposed, such as distributed controllers [5–10]. To the best of our knowledge, the most powerful dynamic security architecture for SDN is the Mcad-SA proposed in [9] which exploits heterogeneity, redundancy, and dynamism from multiple controllers to intensify security. However, no researches about their security performance evaluation have been conducted.

Our objective is to develop a general model to evaluate effectiveness of dynamic SDN architectures (take Mcad-SA as the instance) and provide some insights for designers to devise new, simple, and effective frameworks. To improve applicability, the model must have the ability to represent features of dynamic frameworks. Besides, it can capture the essential dynamic of progressive attack interacting with defense strategies to hinder that progression [11].

In this paper, we examine an abstract SDN-defense scenario designed to simulate strategic interactions between attacks and defense methods in dynamic architectures, as the control layer is a critical part in SDN and responsible for handling and distributing flows of information between network applications and the data plane. Thus we take the control plane's security as measurement of the whole SDN. Though simple, our model is generic, flexible, and applicable for a range of dynamic architectures and defense techniques. Our approach is game-theoretic which allows us to realize how effective are dynamic architectures when they deal with attacks and how security performance varies

as attackers and defenders change their behaviors. Unlike other game-theoretic researches, our model can apply to dynamic systems. It can also imitate systems which adopt specific defense means, like moving-target defense. And via systematic simulation, this empirical approach provides us with the opportunity to quantitatively estimate security performance of dynamic systems and defense technologies.

Among our findings, we characterize the control plane's security as the contention of controllers in it by opponents. And controllers' compromising process is proceeding procedurally, which means the probability of controllers being compromised is a function of the number of probes implemented on them. Besides, defenders have the capability to perceive the secure states of controllers since some dynamic architectures own detection mechanisms. Through experimental simulations, we observe that Mcad-SA are powerful against persistent probes from attackers. And reasonable defense and scheduling mechanism can further improve SDN's security. Besides, it is fascinating that system's detection capability is not as important as it is imagined in Mcad-SA since dynamism and redundancy can mitigate this weakness to a large degree.

The paper is organized as follows. The next section describes related work. Section 3 presents a detailed specification of our games. In Section 4, we present our game-theoretic experimental results. The last section concludes by summarizing our work and discussing future work.

2. Related Work

Recently, researchers have paid attention to security in SDN. On one hand, it aims at designing more resilient and robust controllers, such as FortNOX [12] and SE-Floodlight [13], are designed to deal with flow-rule conflict [14]. On the other hand, many SDN architectures with distributed controllers have been proposed to intensify SDN security from the point of framework. However, little work has been done on evaluating their security performance, especially when the architecture adopts moving-target defense to enhance security.

However, there is an amount of literature on computer security. In [15], formal methods have been used to provide an attack surface metric as an indicator of the system's security. So reducing attack surface is an effective approach to mitigate risk. Furthermore, [16, 17] pointed out launching a sequence of attacks is a common way for attackers to exploit vulnerabilities at multiple stages of the system. Thus, dividing the system into several layers and using attack graphs is a feasible method to assess the cause-consequence relationships between diverse network states. And it is a popular trend that proactive defense techniques are employed by the system to generate security strategies that alter over time to reduce the exposure of vulnerabilities and increase complexity and attack costs [18, 19]. Against such defense mechanisms, game theory provides an appropriate theoretical framework for modeling the win-lose situation between a defender and attacker [20]. In [21], FlipIt game, where two players compete for control of single resource, is designed to describe uncertainty in state of control. Extensions of FlipIt have considered additional actual scenario features. In an extension called "FlipThem" [22], multiple servers are incorporated and authors consider some extreme situations where attackers have to compromise one or all servers to achieve goals. Reference [11] refines the scenario further. It devises a more flexible utility model to allow payoffs between objectives of control and availability. Meantime it considers imperfect probe detection which indicates the defender observes attack probe actions with probabilities. Moreover it provides a much richer set of attack and defense strategies to evaluate overall system state. Although above work is related to security analysis, it cannot be applied in SDN environment directly, because SDN is a novel framework and has its own traits. For instance, its primary goal is to guarantee validity of flow rules delivered to switches. Besides, its defense strategy is backup or switch instead of reimage. Thus, we further make improvements on existing models and apply them into the SDN scenarios.

3. Game Specification

In our game the attacker and defender compete for the control of M controllers. Because once controllers are compromised then the whole network is also manipulated by attackers. M is changeable since the number of running controllers is varying with time in Mcad-SA. At first, controllers are in control of defenders. The attacker attempts to wrest control of a controller through via inserting some malicious flow rules, which needs probe actions on controllers. The success probability is relevant to the number of probes. Meantime, the defender may take some defense strategies based on current SDN frameworks. For instance, it may switch to another backup controller after it discovers the controller runs abnormally. Based on above analysis, first we introduce the abstracts of Mcad-SA and controller and then present models of players in detail later.

3.1. Abstract of Mcad-SA. In this section, we attempt to represent the control plane in Mcad-SA with a model. The reason why we take Mcad-SA as the example is that it is a typical representation of dynamic SDN architecture which employs heterogeneity, redundancy, and dynamism to improve security and owns perception simultaneously. The overview of Mcad-SA is presented in Figure 1. It consists of a variant control plane, a sensor, a scheduler, and an arbitrator. The novel control plane is equipped with multiple controllers. And the scheduler will select several as running controllers via specific mechanisms. In the meantime, the sensor can perceive current state of each controller to some extent. For example, it has the ability to realize how many probes have been conducted on controllers by attackers. As to the arbitrator, it determines the valid and correct flow rules down to switches.

The basic elements of the control plane in Mcad-SA include the number of controllers, the running controllers set each time, whether it has perfect perception, and what

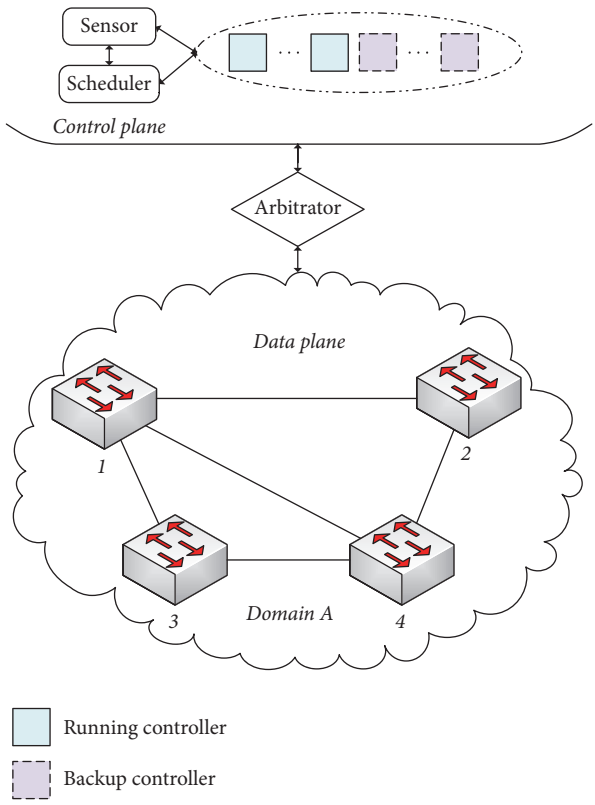


Figure 1: An overview of Mcad-SA.

scheduling strategies it adopts. Formally, abstract model of the control plane is a quadruple $\langle c, r, \psi, \Omega \rangle$, where

(i) $c \in [1\ N]$ represents how many controllers the control plane has and N is a very big positive number;

(ii) $r \in [1\ c]$ represents number of running controllers; r is changeable and generally an odd number;

(iii) $\psi \in \{0, 1\}$ indicates whether the control plane can sense the states of all controllers ideally; $\psi = 1$ indicates it can detect every probe the attackers implement, while $\psi = 0$ means it detects probe with some probability, and the probability is related to number of probes already on controllers;

(iv) Ω illustrates sets of strategies the scheduler employs, such as random selection or other scheduling algorithms.

Generally, there are $2|\Omega|$ occasions according to above settings. For instance, when $\psi = 1$ and Ω is random selection, which indicates the system can be aware of every probe precisely and it chooses controllers randomly as the next running controller set. Next, we introduce the model of players.

3.2. Model of Player. In order to quantitatively measure security performance of the control plane, we need define information of players. Here we focus on their actions, utilities, and strategies.

3.2.1. Actions. Before describing players' actions, we depict the state of a controller at first, which is convenient for modeling players. At any point of time, controllers' state can be represented by who takes control of it, what type, whether it is up or down, and how much progress the attacker has made towards controlling if the attacker has begun his probe action. Formally, controller state is a triple $\langle \chi, \kappa, s, \rho \rangle$, where

(i) $\chi \in \{\text{att}, \text{def}\}$ represents the player who controls the controller;

(ii) κ represents what kind of controller it belongs to, such as floodlight [23], Ryu [24], or OpenDaylight [25].

(iii) $s \in \{\text{up}\} \cup \{\text{down}\}$ represents whether the controller is running ($s = \text{up}$) or is not up ($s = \text{down}$);

(iv) ρ is the successful number of probes attackers has carried out on a controller.

The state of the overall SDN system is determined by the joint state of all running controllers, plus the current time t.

For attackers, its action is called *probe*. To describe the actions precisely, let $\langle \chi_t, \kappa_t, s_t, \rho_t \rangle$ be the state at time t. And we denote $\langle \chi_{t+}, \kappa_{t+}, s_{t+}, \rho_{t+} \rangle$ as the state after the actions. Thus the probe action's effect can be specified via the following rules:

(i) If $s_t = \text{down}$, the probe action has no effect: $\langle \chi_{t+}, \kappa_{t+}, s_{t+}, \rho_{t+} \rangle = \langle \chi_t, \kappa_t, s_t, \rho_t \rangle$.

(ii) If $s_t = \text{up}$ and $\kappa_{t+} = \kappa_t$, the number of probes is incremented: $\rho_{t+} = \rho_t + 1$, while $s_t = \text{up}$ and $\kappa_{t+} \neq \kappa_t$, which indicates, during the probe, the running controller has been replaced with another type of controller. In Mcad-SA, this switching mechanism can intensify the security of control plane to some degree since they have different bugs, which requires more cost to attack successfully. So the number of probes is incremented with probability $1-e^{-\beta\rho_{t+}}$ under this occasion because formal effective probe may be invalid on another type of controller. And when $\chi_t = \text{att}$, then the attacker maintains control: $\chi_{t+} = \text{att}$. While $\chi_t = \text{def}$, the defender continues to control the controller with probability $e^{-\gamma\rho_{t+}}$. For attackers, its probability of control raises up to $1 - e^{-\gamma\rho_{t+}}$. This probability can be seen as the probability with which the controller generates malicious flow rules, where $\beta > 0$ and $\gamma > 0$ are scaling parameters.

The defender has two actions. One is reimage and the other is switch. The goal of reimaging controllers is reset its state to initial settings. So the defender wins control back and the cumulative effect of probe is eliminated. And the state is reset as follows: $\langle \chi_{t+}, \kappa_{t+}, s_{t+}, \rho_{t+} \rangle = \langle \text{def}, \kappa_t, \text{down}, 0 \rangle$. The switch action is to replace the running controller with another controller, which can reduce the probability of control by attackers to some extent. And the state is transited to that of new controller: $\langle \chi_{t+}, \kappa_{t+}, s_{t+}, \rho_{t+} \rangle = \langle \chi_t', \kappa_t', s_t', \rho_t' \rangle$.

3.2.2. Utility. In SDN, the major function of controllers is producing valid flow rules and delivering them to switches.

And the purpose of the defender is to guarantee the control plane generates as many right flow rules as possible, while the attacker' objective is just the opposite. Thus we measure each player's payoff based on the state of flow rules the control plane creates.

We presume the number of effective flow rules a healthy controller produces is n_e^h. If s_t = down, the controller cannot generate any valid rules. While if s_t = up, the number of valid rules n_e^p is related to probes the attacker conducts (i.e., ρ_t). It can be calculated via

$$n_e^p = n_e^h e^{-\gamma\rho_t}. \tag{1}$$

Then we can define each player's payoff. Given the running controller set C_R, the total payoff of the attacker is

$$\mathrm{AP}_{\mathrm{total}} = \sum_{c_i \in C_R} n_e^h \left(1 - e^{-\gamma\rho_t}\right) - \Phi_{\mathrm{ac}}, \tag{2}$$

where c_i is the ith controller and Φ_{ac} is the total attacking cost and proportional to the number of probes. Correspondingly, the defender's payoff can be computed via

$$\mathrm{DP}_{\mathrm{total}} = \sum_{c_i \in C_R} n_e^p - \Phi_{\mathrm{dc}}, \tag{3}$$

where Φ_{dc} is the total defending cost. It includes the cost of reimaging and switching to other controllers and is proportional to the number of reimages and switches.

3.2.3. Strategies. A strategy for players is to choose controllers on which they execute their actions. They are always triggered by observed events or detected information.

The attacker's strategy consists of two factors: how many probes they can execute each time and the number of probes they have conducted on each controller. Here we consider a strategy defined by the following policy rules:

(i) *Random-Probe (RanProb):* select uniformly among running controllers under defender's control (χ_t = def).

(ii) *Maximum-Probe (MaxProb):* select controllers that have been probed most to proceed next probe.

For the defender, the condition of executing reimage is similar to attackers. Also it has the following strategies:

(i) *Random-Reimage (RanReimage):* pick up running controllers randomly to reimage regardless of their states.

(ii) *Maximum-Reimage (MaxReimage):* choose controllers that have been probed most to be reimaged.

As to the other action of defender, the switch action is executed under two mechanisms. One is after a fixed time interval Δt, regardless of whatever states all running controllers are. The other is according to the fraction of controllers regulated by defenders. When it falls below a threshold τ, the switching process is evoked. However, when $\psi = 0$ the defender cannot realize controllers' state directly. Instead, the sensor component can assist the control plane estimating their conditions. Let n_c^{def} denote the estimated number of controllers in the hand of defenders; if

$$\frac{\mathbb{E}\left[n_c^{\mathrm{def}}\right]}{|C_R|} < \tau, \tag{4}$$

then the switch action is activated. One detail to be mentioned, the switch strategy may have more than one method according to system settings. Here, we make specific explanations on switch strategies (Ω). We list three common kinds of scheduling methods below. And we let C_n be the next set of running controllers.

(i) *RandomWithRepeat:* select C_n uniformly from all the available controllers.

(ii) *RandomWithoutRepeat:* select C_n uniformly from the rest controllers to guarantee $C_n \cap C_R = \emptyset$.

(iii) *MaxSG*: this is a perceptive scheduling strategy proposed in [26] to maximize security gain of the control plane during each switch. For a controller, it will be switched preferentially to another controller which is belonged to another type, deployed on another host and probe least. That is to say, this mechanism ensures two controllers have maximum difference and the next running controller is the most reliable one.

4. Game-Theoretic Analysis

In this part, we conduct simulations on Mcad-SA based on above assumptions. First, we present a brief introduction of its operating mechanisms again.

In Mcad-SA, the running controller set is varying with time under this situation. Moreover, the system can adjust its switching strategy based on specific settings. That illustrates that dynamism is introduced into SDN.

4.1. Simulation Environments. It is obvious that this game is a periodic process. So we let this game last for T time units (T = 200). For the control plane employing more than one controller, we take M = 7. The scaling parameters β and γ equal 0.1. The fixed switch interval Δt is set as 25 and the reimage interval is 20. And we let τ equal 0.5.

Next, we implement experiments using a discrete-event simulator. Before the simulation, we define f_p as failure probability of the control plane. f_p is related to conditions of all running controllers. Only when over $\lceil(|C_R| + 1)/2\rceil$ controllers are compromised simultaneously will the system fail. For controller $c_i \in C_R$, its reliability r_i^c equals the ratio of valid flow rules (n_e^p) to total rules it produces (n_e^h). Then f_p can be computed via (5), where $C_{c\geq\lceil(|C_R|+1)/2\rceil}$ represents the set of all situations with more than $\lceil(|C_R| + 1)/2\rceil$ controllers being compromised simultaneously. C_c^u is the set of benign controllers in C_c. The smaller f_p is, the more right flow rules the control plane can produce, which means more powerful ability to resist attacks. Here, we let n_e^h be 1000:

$$f_p = 1 - \sum_{C_c \in C_{c\geq\lceil(|C_R|+1)/2\rceil}} \prod_{c_i \in C_c^u} \left(1 - r_i^c\right) \prod_{c_j \in C_c \setminus C_c^u} r_j^c. \tag{5}$$

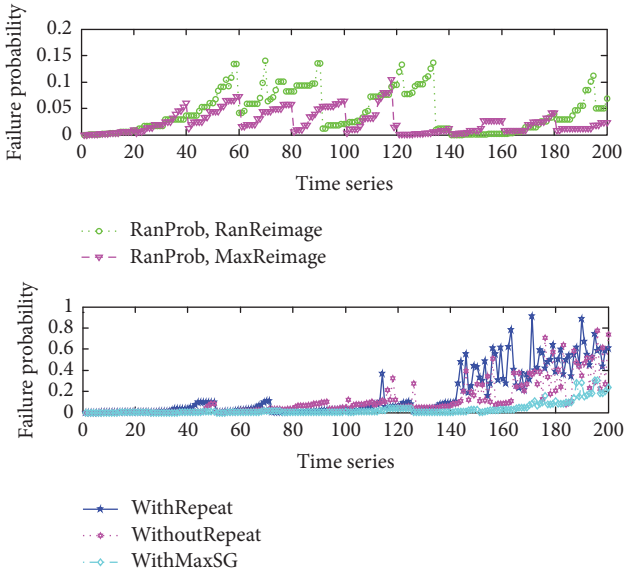


Figure 2: Failure probability under respective defending strategies when facing random probes.

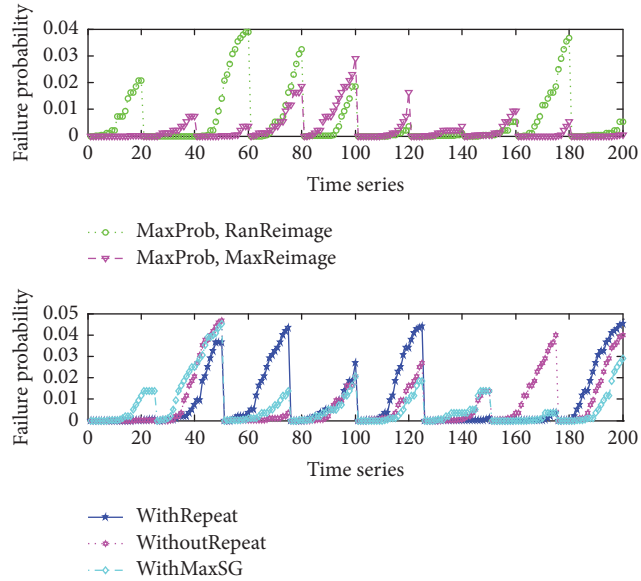


Figure 3: Failure probability under respective defending strategies when facing maximum probes.

4.2. Simulation Process. At first, we assume that all controllers are perfect without any probes. Thus, f_p is 0 at the beginning via (1). Then attackers will choose an attacking strategy to conduct attacks, which results in the rise of f_p. With f_p being higher and higher, the system is going to launch respective defense mechanisms to reduce f_p. As a result, f_p is varying with time and players' strategies. Through analyzing the changing trend of f_p which is related to status of running controllers, we can realize the real-time state of Mcad-SA under various combinations of players' tactics. Similarly, players' payoffs can be obtained according to probes on controllers and cost of players via (2) and (3). Then we can conclude that which player owns the control of the system.

Further, in order to measure how powerful Mcad-SA is, we classify the scenarios of Mcad-SA into two situations according to ψ. When $\psi = 1$, we call it perfect probe detection environment, while $\psi = 0$, it is regarded as imperfect probe detection environment. Then we analyze the simulation results under two circumstances when adopting various scheduling strategies (Ω).

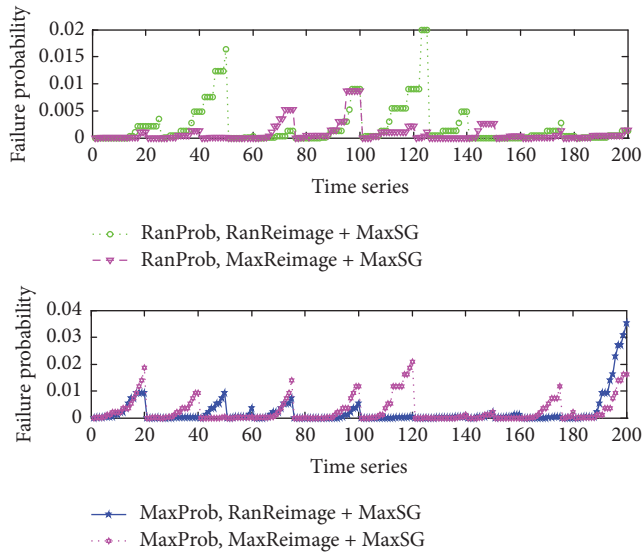


Figure 4: Failure probability with associate defending strategies under perfect detections.

4.3. Simulation Results

4.3.1. Perfect Probe Detection Environment. In perfect probe detection environment, the defender is able to observe all probes from attackers on controllers.

(a) Respective Defending. First, we make comparisons on the effectiveness of respective defending strategies when facing different attacking means. Figure 2 indicates variations of system's security performance when attackers adopt random probe, while Figure 3 illustrates the situations when attackers employ maximum probes.

In Figure 2, the upper picture is the result when taking reimage actions only while the map below is the consequence when taking switch actions only. It is obvious that when the defender adopts reimage methods to defend random probes, random and maximum reimages have similar effect (MaxReimage is slightly better). However, the consequences of various switching strategies are very different. When the system employs random switching (with repeat or without repeat), security performance is close. While adopting *MaxSG*, reliability of the system is strengthened to some degree, especially after 100 time units. The reason is that probes of controllers will increase to a large number with time going on. So under this situation, perceptive switch can ensure the system select more reliable controllers (probed least) than random switch. This phenomenon further demonstrates superior scheduling ways have better effect on improving system's security performance.

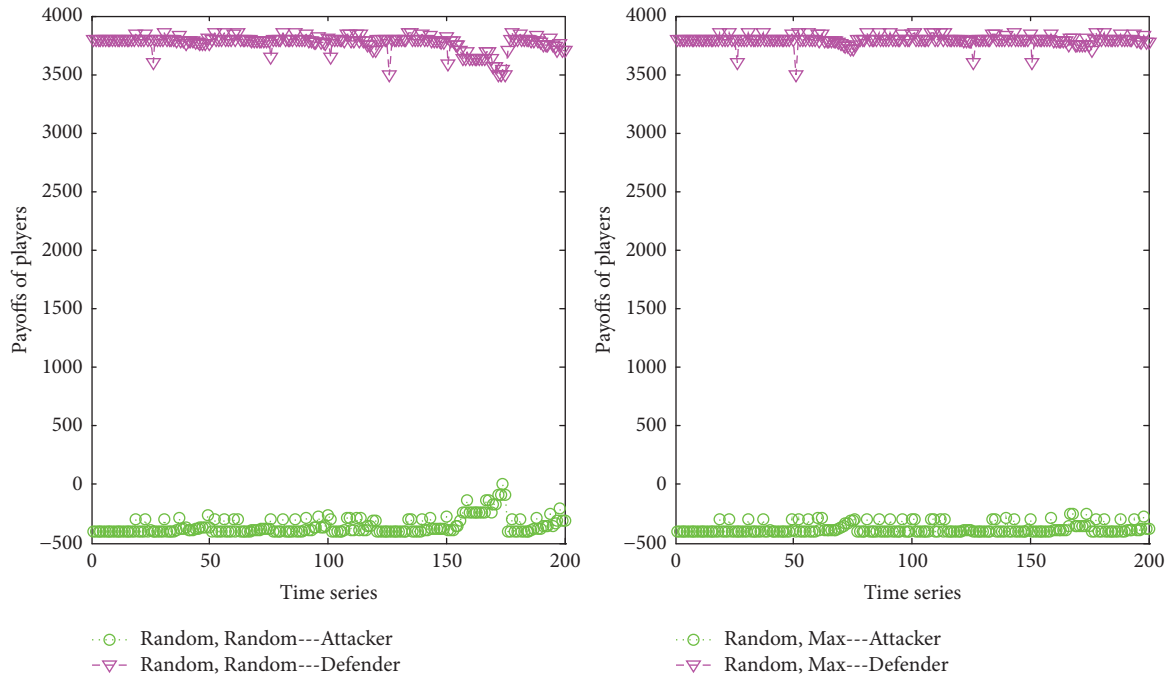


Figure 5: Payoffs of players against random probes with associate defending strategies under perfect detections.

In Figure 3, it is another situation where the attacker conducts maximum probes. An interesting phenomenon is observed that all defending measures have approximate preventive effect. That is due to the cause that this attack mode results in severe destruction on specific controllers. Thus once defending methods can avoid selecting these controllers, the security state of the system can be guaranteed. So it comes to a conclusion that unless attackers have to probe particular controllers to acquire important information, it is of little value to carry out persistent maximum probes.

(b) Associate Defending. As is analyzed above, *MaxSG* is the most effective switch option compared to random switch. Then we combined it with reimage action to testify associate defending performance. That is to say, the system adopts reimage and switch behaviors simultaneously.

Figure 4 illustrates the results of associate defending when encountering random and maximum probe individually. The upper section is the consequence against random probe. It is apparent that associate defending mechanism significantly reduces the failure probability of the system and enhances its security gain as the value is decreased with an order of magnitude (from 0.05 to 0.005 averagely), while the bottom part is the case against maximum probe. Similarly, security of the system is intensified to some extent and the value of failure probability is lowered to half of that in previous circumstance.

(c) Players' Payoffs. Next, we analyze players' payoffs in occasions where associate defending is used to deal with probes from attackers. Figure 5 is the case with random probe while Figure 6 is the situation with maximum probe.

In Figure 5, the left part shows how payoffs vary with random reimage and *MaxSG* while the right section is the result with maximum reimage. Both pictures manifest that payoffs of players are steady, which means attackers cannot acquire much valuable information from the system and defenders can maintain it operating in a relatively secure state meantime. As to some downward pulse, their appearances are due to the trigger of reimage or switch actions. However, strictly speaking, defense effect of the right one is superior to that of the left one since the line of the attacker is more stable and the value is lower, which further proves maximum reimage is better than random reimage under this environment.

Figure 6 is similar to the shape of Figure 5 except that downward pulses are more obvious. That is an indication that the system is severely compromised since several controllers have been probed persistently under maximum probes. But in other occasions, the payoff of the attacker is less than that in Figure 5, which again demonstrates maximum probe is not a good choice unless attackers have special purposes.

Actually, the described process can be regarded as an approximate zero-sum game. Once the attacker gains some privilege, the defender loses some control and vice versa. Based on above simulations and analysis, the following conclusion can be drawn: introducing dynamism to SDN is a feasible and effective way to mitigate attacks' impact and improve the system's robustness. And whatever actions attackers take, the best response for defenders is maximum reimage and *MaxSG*. In other words, eliminating and restoring the most damaged components in the system is a critical point to ensure its security.

4.3.2. Imperfect Probe Detection Environment. Then we analyze the results when the system does not own the capability to detect each probe ($\psi = 0$, i.e., imperfect detection).

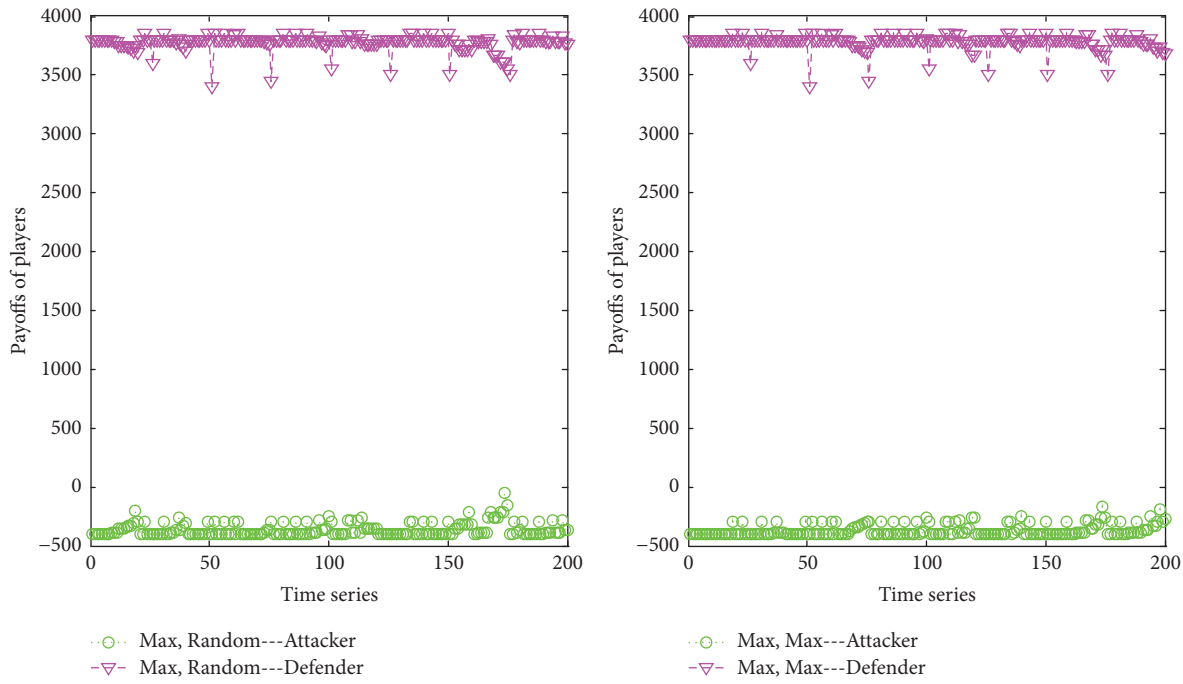


Figure 6: Payoffs of players against maximum probes with associate defending strategies under perfect detections.

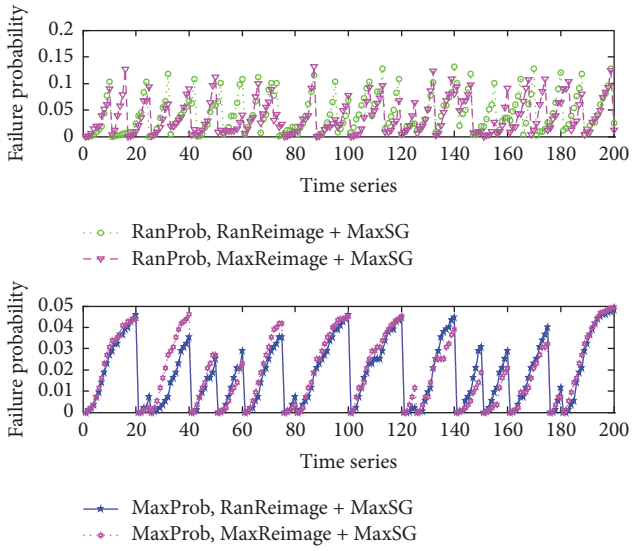


Figure 7: Failure probability with associate defending strategies under imperfect detections.

Here, we neglect the analysis on respective defending since its changing trend is similar to that in perfect probe detection environment. So we pay attention to associate defending and players' payoffs.

(a) Associate Defending. Figure 7 is the simulation under imperfect probes. Compared to perfect probe, the most distinction is the failure probability increases overall. Actually, the system's security performance under two associate defenses is approximate. The cause resulting in this phenomenon is that the system can no longer perceive the real state of controllers, which may incur the scheduler to make an incorrect policy. For instance, the scheduler seems to choose a controller which has been probed least to be the next running controller, but actually this one has already been severely compromised. This phenomenon may also appear in the reimage situations. Above decisions will influence the system's reliability to some extent. However, a point to be noticed, although without perfect probe detection, the failure probability is also maintained at a low level because of reimage and switch strategies.

(b) Players' Payoffs. Figures 8 and 9 are the variations of players' payoffs under random and maximum probes, respectively. The distribution of payoff is fluctuating extensively, particularly in random probe. That is due to the reason that defenders are likely to discover the controllers being continuously compromised when attackers employ maximum probes even under imperfect detections. Thus, we can notice downward pulses are more distinct in Figure 9 than Figure 8, which is strong evidence on the effectiveness of reimage and switch strategies in maximum probes. Meanwhile, the attacker's payoff is clearly increased in both figures. Nevertheless, it is still kept in a stable interval, which demonstrates players reach an equilibrium. Moreover, for attackers, the results further authenticate that adopting random probe is superior to maximum probe in general.

According to above analysis, it is evident that detection capability has effect on the security performance of the system to some extent since inaccurate state information of controllers will definitely influence the decision-making process of the scheduler. And ways of reimage and switch

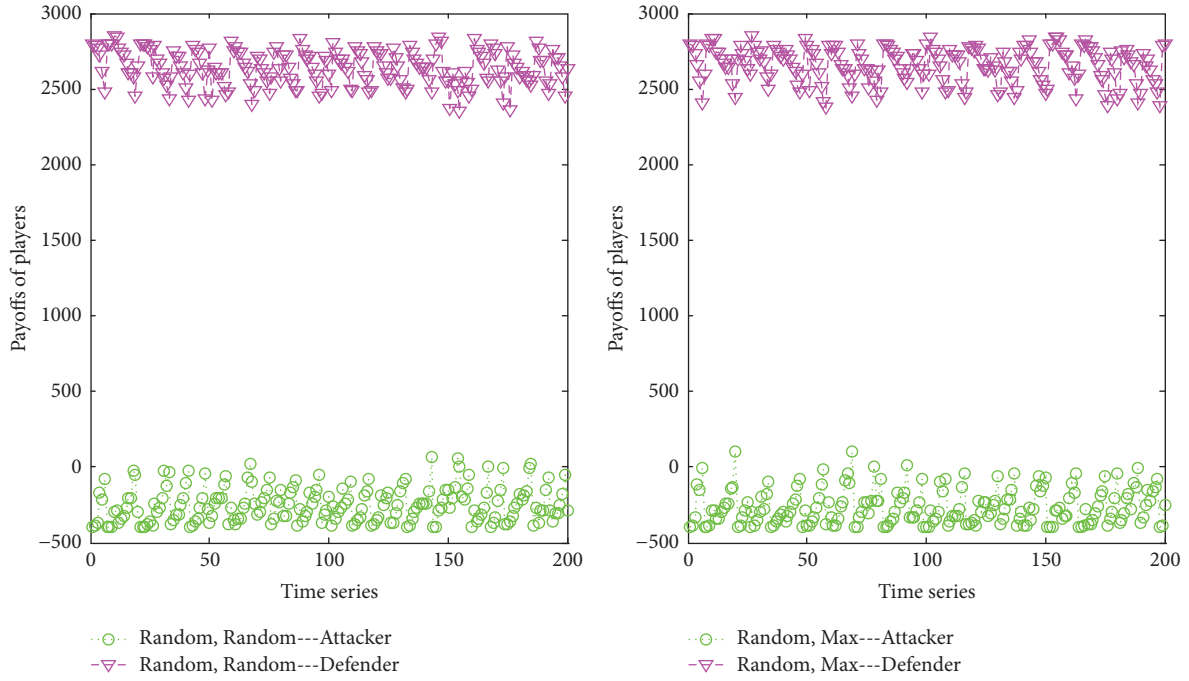


Figure 8: Payoffs of players against random probes with associate defending strategies under imperfect detections.

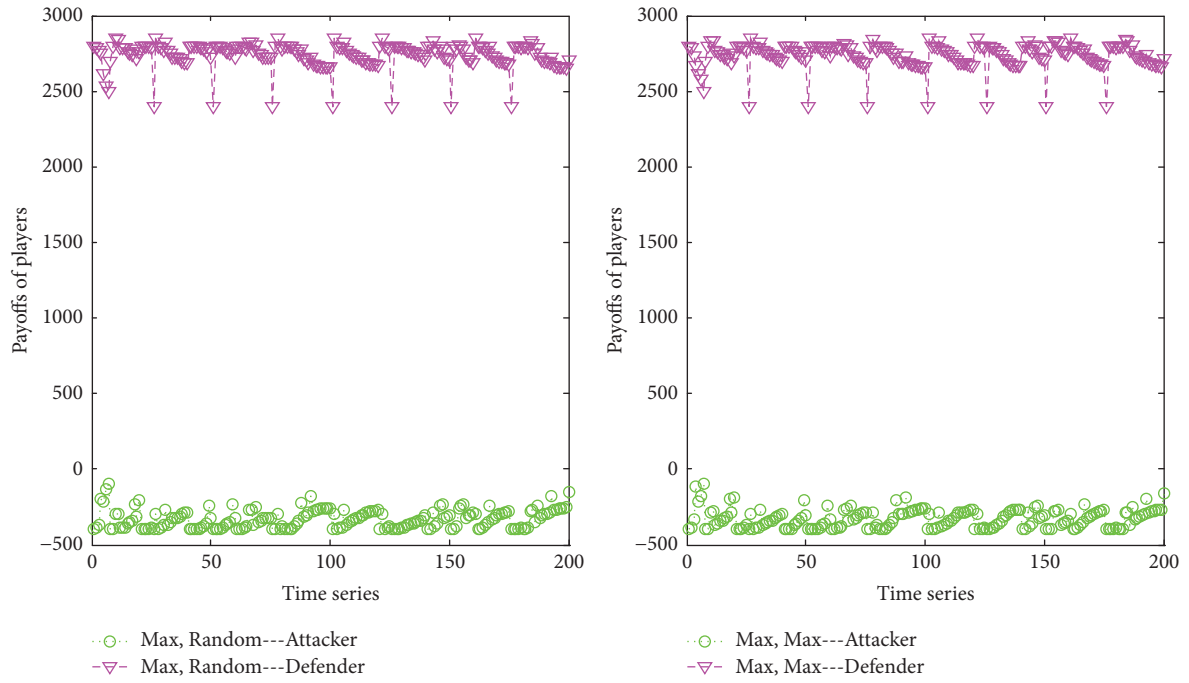


Figure 9: Payoffs of players against maximum probes with associate defending strategies under imperfect detections.

can lead to diverse defense results. In most cases, random probe is a better option for attackers while the best response of defenders is the combination of maximum reimage and *MaxSG*. However, as long as dynamism, heterogeneity, and redundancy are introduced into SDN, its security can be improved significantly and the failure probability of the control plane can be maintained at relatively a low standard.

4.4. Equilibrium Results. In this section, we report the equilibria found in the simulation. As we all know, when the system reaches an equilibrium, the failure probability of Mcad-SA and players' payoffs stays stable. They have no intent to alter their strategies.

Actually in Mcad-SA, whenever in perfect or imperfect probe detection environment, the situations of equilibria are

identical. The only occasion in which the equilibrium exists is that defenders adopt maximum reimage and MaxSG scheduling method while attackers select maximum probe. From the simulation results above, as long as attackers carry out maximum probe, they always can acquire higher payoffs and higher failure probability of the system over random probe on matter what actions defenders take. Similarly, defenders will tend to choose maximum reimage and MaxSG to obtain higher payoffs and lower failure probability whenever they face random or maximum probe. In other words, maximum reimage and MaxSG are the best choice for defenders and maximum probe is the best option for attackers as they can achieve their goals to maximum extent.

5. Conclusion

Evaluation of security performance of SDN architectures plays a critical role in designing reliable structures and estimating system risks. Focused on Mcad-SA, a novel SDN structure, we attempt to establish a model to analyze its ability to resist attacks with the assistance of game theory. This model can represent players' related information (actions, strategies, etc.) and quantitatively assess system's capability to deal with risks. Experimental results indicate the introduction of dynamism, redundancy, and heterogeneity into SDN can intensify system's security and maintain its consistent capability against diverse probes which is an extraordinary weakness in current SDN frameworks. Further, we analyze equilibrium in particular situations where common types of probes and defense methods are included. And analysis results present hints that for dynamic systems the design of their defense strategies has certain effect on security enhancement, which implies that how to devise effective defense mechanism is crucial to maximize security gain. In the future, we plan to take internal security mechanism of diverse controllers into consideration and pay attention to hybrid security analysis.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (no. 61521003), the National Key R&D Program of China (no. 2016YFB0800100 and no. 2016YFB0800101), and the National Natural Science Foundation of China (no. 61602509).

References

[1] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] D. He, S. Chan, and M. Guizani, "Securing software defined wireless networks," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 20–25, 2016.

[3] C. Monsanto, J. Reich, and N. Foster, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pp. 1–14, USENIX Association, 2013.

[4] Z. Guo, M. Su, Y. Xu et al., "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Computer Networks*, vol. 68, pp. 95–109, 2014.

[5] P. Berde, M. Gerola, J. Hart et al., "ONOS: towards an open, distributed SDN OS," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*, pp. 1–6, August 2014.

[6] V. Yazici, M. O. Sunay, and A. O. Ercan, "Controlling a software defined network via distributed controllers," in *Proceedings of the 2012 NEM Summit*, pp. 16–20, 2014.

[7] H. Li, P. Li, S. Guo, and A. Nayak, "Byzantine-resilient secure software-defined networks with multiple controllers in cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 4, pp. 436–447, 2014.

[8] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: a compositional hypervisor for software-defined networks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pp. 87–101, USENIX Association, May 2015.

[9] C. Qi, J. Wu, H. Hu et al., "An intensive security architecture with multi-controller for SDN," in *Proceedings of the 35th IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS '16)*, pp. 401-402, April 2016.

[10] K. ElDefrawy and T. Kaczmarek, "Byzantine fault tolerant software-defined networking (SDN) controllers," in *Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC '16)*, pp. 208–213, June 2016.

[11] A. Prakash and M. P. Wellman, "Empirical game-theoretic analysis for moving target defense," in *Proceedings of the 2nd ACM Workshop on Moving Target Defense (MTD '15)*, pp. 57–65, 2015.

[12] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*, pp. 121–126, August 2012.

[13] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software Defined Network Control Layer," in *Proceedings of the Network and Distributed System Security Symposium (NDSS '15)*, San Diego, CA, USA, 2015.

[14] Z. Guo, Y. Xu, M. Cello et al., "JumpFlow: Reducing flow table usage in software-defined networks," *Computer Networks*, vol. 92, pp. 300–315, 2015.

[15] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011.

[16] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using Bayesian attack graphs," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 61–74, 2012.

[17] C.-C. Ten, C.-C. Liu, and M. Govindarasu, "Vulnerability assessment of cybersecurity for SCADA systems using attack trees," in *Proceedings of the IEEE Power Engineering Society General Meeting (PES '07)*, pp. 1–8, Tampa, FL, USA, June 2007.

[18] S. Jajodia, A. K. Ghosh, and V. Swarup, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, Springer Ebooks, 2011.

[19] S. Jajodia, S. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense II: Application of Game Theory and Adversarial Modeling, Advances in Information Security*, Springer, 2012.

[20] M. H. Manshaei, Q. Zhu, T. Alpcan, T. Basar, and J.-P. Hubaux, "Game theory meets network security and privacy," *ACM Computing Surveys*, vol. 45, no. 3, article 25, 2013.

[21] M. Van Dijk, A. Juels, A. Oprea, and R. L. Rivest, "FlipIt: The game of "stealthy takeover"," *Journal of Cryptology*, vol. 26, no. 4, pp. 655–713, 2013.

[22] A. Laszka, G. Horvath, M. Felegyhazi, and L. Buttyán, "FlipThem: modeling targeted attacks with flipit for multiple resources," in *Decision and Game Theory for Security*, vol. 8840 of *Lecture Notes in Computer Science*, pp. 175–194, Springer International Publishing, Cham, 2014.

[23] FloodLight, "Open SDN controller," http://floodlight.openflowhub.org.

[24] "Ryu SDN Framework," http://osrg.github.io/ryu.

[25] "OpenDaylight Consortium," http://www.opendaylight.org.

[26] C. Qi, J. Wu, H. Hu, and G. Cheng, "Dynamic-scheduling mechanism of controllers based on security policy in software-defined network," *IEEE Electronics Letters*, vol. 52, no. 23, pp. 1918–1920, 2016.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 6751042, 13 pages
https://doi.org/10.1155/2018/6751042

Research Article

Duo: Software Defined Intrusion Tolerant System Using Dual Cluster

Yongjae Lee, Seunghyeon Lee, Hyunmin Seo, Changhoon Yoon, Seungwon Shin, and Hyunsoo Yoon

KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea

Correspondence should be addressed to Seungwon Shin; claude@kaist.ac.kr

Received 30 September 2017; Accepted 18 December 2017; Published 21 January 2018

Academic Editor: Qiang Fu

An intrusion tolerant system (ITS) is a network security system that is composed of redundant virtual servers that are online only in a short time window, called exposure time. The servers are periodically recovered to their clean state, and any infected servers are refreshed again, so attackers have insufficient time to succeed in breaking into the servers. However, there is a conflicting interest in determining exposure time, short for security and long for performance. In other words, the short exposure time can increase security but requires more servers to run in order to process requests in a timely manner. In this paper, we propose Duo, an ITS incorporated in SDN, which can reduce exposure time without consuming computing resources. In Duo, there are two types of servers: some servers with long exposure time (White server) and others with short exposure time (Gray server). Then, Duo classifies traffic into benign and suspicious with the help of SDN/NFV technology that also allows dynamically forwarding the classified traffic to White and Gray servers, respectively, based on the classification result. By reducing exposure time of a set of servers, Duo can decrease exposure time on average. We have implemented the prototype of Duo and evaluated its performance in a realistic environment.

1. Introduction

Nowadays, (nearly) everything in our lives is digitalized and connected to each other, and it makes us easily access necessary information. However, the richness of the connectivity causes another side effect that motivates cybercriminals to reach connected resources for malicious purposes. To minimize this effect, researchers and practitioners have been putting huge amount of effort into devising diverse network security approaches. Those approaches are commonly classified into two types: (i) signature-based network attack detection system (e.g., Snort [1]) and (ii) network anomaly detection system [2]. No one has doubt that they have effectively and efficiently protected our network environments so far.

However, cyberattacks have become more sophisticated and existing network intrusion detection/prevention mechanisms are no longer effective against such advanced attacks [3]. In response to this problem, researchers have proposed network intrusion tolerant system (ITS), which proactively defends victim systems without detecting nor blocking intrusion attempts [4, 5]. Of the various types of ITS proposed until today, recovery-based ITS, which periodically reverts the production services to their clean initial states, is now widely endorsed and deployed [3, 6, 7].

Technically, the recovery-based ITS can effectively reduce attack surfaces of the servers [8] by making the servers available to the public including any potential adversaries for a very short period of time and restoring possibly compromised servers. Such a time period is referred to as *exposure time*, and it plays an extremely crucial role in recovery-based ITS [7, 9, 10]

The exposure time in recovery-based ITS creates a trade-off between the cost and security. Taking the short exposure time leads a server to be more intrusion tolerant, but it will require more backup servers for high service availability [7]. In contrast, the longer exposure time will require less computing resources, accordingly. If the exposure time is

configured to be too long, it will increase the chances of the server being compromised. To this end, finding the optimal exposure time for a recovery-based ITS is an important problem.

Despite its importance, the matter of finding the optimal exposure time is understudied and still remains unanswered. The previous studies have only focused on reducing attack surfaces of target servers [8, 11] or efficiently recovering target servers [12, 13]. Indeed, they have pointed out that finding the best exposure time for ITS is a critical problem that must be solved, but this problem has never been clearly addressed.

In this paper, we propose Duo, a recovery-based ITS that leverages software defined networking (SDN) and network function virtualization (NFV) technologies to minimize the overall exposure time without consuming additional computing resources. Unlike the previous ITS proposals that applied uniform exposure time for all the servers, Duo adaptively controls the exposure time. To do this, Duo first classifies network requests into two criteria, *benign* and *suspicious*, using the existing network security systems (e.g., NIDS). The servers on the network are also classified into two server groups, *White and Gray*, and the servers that belong to the White group only handle the benign requests, while the servers in the Gray group handle the rest. Here, the Gray group in Duo is assigned shorter exposure time than that of the White group because it is more likely that the Gray servers will be compromised, and thus the servers are more frequently reverted to their initial state for security purposes.

Although Duo can reduce the systemwide exposure time, it introduces new challenges that must be solved. One challenge is that the traffic classification points become the network bottlenecks. Since our system investigates all the incoming network packets, the network will likely be unstable or even unavailable whenever there is a burst of inbound network traffic. Another challenge is that it should be able to determine how many and which type of servers (Gray or White) should be spawned or killed based on the volumes of suspicious and benign traffic on the network.

In this work, we address the first challenge with the help of SDN [14, 15] that is inherently designed to control network flows flexibly. Duo forwards traffic to a corresponding server according to its type. In SDN, this flow control is easily described in a flow rule; once the rule is activated, successive network flow will be forwarded to appropriate servers. Furthermore, in resolving the bottleneck problem of traffic classification, we employ NFV technology that allows deploying multiple traffic classifiers as a virtual appliance in a distributed manner. In addition, to resolve the second challenge, we propose a novel optimized resource management algorithm that is based on integer linear programming (ILP). When compared to the heuristic server provisioning scheme proposed in [12], our algorithm finds the optimal number and combination of different types of servers within the budget instead of relying on the optimistic assumption that limitless servers are available.

The main contributions of the paper are as follows: (1) we propose a new recovery-based ITS architecture that can reduce the average exposure time by classifying network flows into two types and treating them differently according to their type. (2) To handle each type of network flows differently and to minimize the overhead incurred by the classification process, we introduce SDN/NFV into ITS. To the best of our knowledge, Duo is the first attempt to incorporate ITS into SDN/NFV technologies. (3) We implement a prototype system to evaluate the performance of Duo, and the evaluation result shows that Duo effectively reduces the average exposure time.

The rest of this paper is organized as follows: Section 2 presents preliminaries of ITS and SDN, and related work is given in Section 3. Section 4 describes the architecture of Duo and explains how it works in detail. The performance of Duo is evaluated in Section 5, and we discuss its limitation in Section 6. Finally, Section 7 presents future work and concludes this paper.

2. Background

2.1. Intrusion Tolerant System. Intrusion tolerant system (ITS) is a new type of security framework that ensures the service availability and the integrity of potential victim systems, which could be affected by any type of intrusion attempts [9, 10, 21]. Unlike conventional security systems such as Intrusion Detection and Prevention Systems (IDPS) whose protection effectiveness solely depends on their intrusion detection capability, ITS takes a different approach that does not rely on the intrusion detection techniques in protecting potential victim systems.

The key idea of ITS is that it continuously restores the potential victim systems to their original pristine condition. By doing so, on the one hand, the victim systems can keep providing services even if they have been compromised, and on the other hand, it can fundamentally minimize any potential collateral damage that may be caused by the compromise. For example, production servers usually stay online for several months or even years without shutting down [7], and thus, in a common APT (Advanced Persistent Threat) attack scenario, they are the most preferred targets to compromise and take control over because the attackers can keep a long-term access to them and perform insider attacks against other assets within the security perimeter. ITS can fundamentally block such an illegitimate long-term access to the internal systems that could put the entire network and asset at risk by continuously reverting the production servers.

Meanwhile, maintaining high service availability is as important as ensuring the system integrity; however, ITS's continuous system restoration strategy significantly impacts the service availability. To solve this problem, ITS maintains a number of server clones for each service, and it uses one copy for providing the actual service at a time while having the rest of the copies as the backups in the original pristine state [7]. When the deployed instance expires and goes through the restoration process, replacing the expired instance, one of the backups is immediately deployed, thus enabling a transparent and smooth service handoff. This feature can be easily implemented using virtualization technology. For instance, virtual machines (VMs) could be leveraged to maintain system clones and handoff the service from one clone to another. In addition, compared with restoring physical

Table 1: Summary table of related work.

Related work	Topic	Description
SCIT [7, 16]	ITS	Initially proposed recovery-based ITS
ACT [12, 13]	ITS	Recovery-based ITS with adaptive cluster scaling
CloudWatcher [17]	SDN/NFV	Network flow inspection framework using SDN technology
QoSE [18]	SDN/NFV	On-demand security service provisioning in an SDN/NFV environment
Bohatei [19]	SDN/NFV	SDN/NFV-based DDoS defense system
Snort [1] Suricata [7]	IDS	Open source network intrusion detection system
HIF [20]	NTC	History-based IP filtering for SIP protection

machines, restoring virtual machines are far more time- and cost-efficient.

2.2. Software Defined Networking. Software defined networking (SDN) is a new networking technology that enables centralized network management. In SDN, the control plane is decoupled from the network devices and placed on a centralized controller, which can maintain a global network view. The controller is often implemented in software, and it is responsible for making decisions on how to deal with network flows, while the data plane simply forwards the packet based on the decisions made by the controller. SDN controllers also implement a network abstraction interface, which allows implementing diverse and innovative network functions into SDN applications. For this reason, SDN networks are known to be dynamic, flexible, and programmable.

Network managers can administer the network system flexibly and dynamically. For example, the managers are able to define a new flow rule that forwards network flow incoming from switch A to switch B. If the SDN controller installs the flow rule on the data plane, then the data plane delivers network packets as instructed in the rule. In the traditional network architectures, this process of rule generation and registration is not possible without additional efforts.

Recently, the characteristics of SDN are investigated to find a new possibility that can enhance security performance. In particular, combined with network function virtualization (NFV) [22], SDN attracts much of the attention of security researchers and network device vendors because the combination achieves successes in defending against network attacks including a DoS (denial of service) attack defense and network anomaly detection [17–19, 23].

2.3. Incorporating ITS into SDN. Since Duo serves network traffic separately, it is a natural choice to incorporate the system into SDN. In other words, to solve the bottleneck problem that can be caused by the traffic classification, we distribute the traffic classifiers within the system network. It would be difficult to control incoming traffic to go through the distributed traffic classifiers in the legacy network environment. In SDN, however, the SDN control plane can generate flow rules to forward incoming traffic to one of the deployed traffic classifiers, which will be selected in the way of minimizing routing overhead.

3. Related Work

We now discuss previous studies that have addressed the challenging issues similar to ours and present summaries of them in Table 1.

3.1. Intrusion Tolerant System. The Self-Cleansing Intrusion Tolerance (SCIT) architecture restores its VM servers in the system to their known pristine or initial state in a periodic fashion [7, 16]. A server stays active for a certain period of time, called exposure window, and the SCIT controller recovers the server to its initial state after the exposure window expires. As a consequence, SCIT makes it impossible for an intrusion to reside in the system longer than the predefined exposure window, and thereby damage caused by the intrusion can be minimized.

An intrusion tolerant system based on adaptive cluster transformation (ACT) attempts to guarantee a high level of system availability by transforming its VM cluster in an adaptive manner [11, 12]. A dynamic cluster expansion/reduction scheme, the main feature of this approach, examines whether the volume of incoming requests increases or decreases. If the volume increases due to an explosion of normal traffic or a DDoS attack, it decides to expand its VM cluster by adding more VMs. On the other hand, if the request volume decreases, it reduces the VM cluster by subtracting VMs from the cluster to save its computing resources and prepare for additional attacks.

These ITS architectures seek to gain safety and availability, but they are not yet comprehensive enough to satisfy both security attributes simultaneously. In other words, SCIT does not account for attacks to exhaust computing resources such as network bandwidth or CPU, which are the major threat to modern network systems. ACT-based ITS, on the other hand, focuses on availability rather than safety, and it is improper for large-scale server systems because a threshold should be calculated in advance to change the cluster size, and it adds or drops only 2 VMs at a time.

3.2. Software Defined Networking. Combination of SDN and NFV technologies contributes to the enhancement of network security systems. CloudWatcher monitors network flows and selects optimal routing paths for network flows to be inspected by security devices [17]. To this end, CloudWatcher's routing selection algorithm is implemented on

top of SDN that has well-defined functionalities to control network flows flexibly.

QoSE is a network security framework that provides security services in an adaptive fashion employing NFV [18]. Specifically, QoSE framework is composed of multiple virtualized network function middleboxes, and considering not only path status but also the middleboxes' availability, QoSE forms paths through which network flows should pass.

Bohatei is a DDoS defense system that is also based on SDN and NFV [19]. Unlike conventional DDoS defense mechanisms, Bohatei virtualized DDoS defense appliances based on NFV and distributed them flexibly. Hence, network flows do not have to go through a certain security middlebox that is fixed at a certain point, which can act as a bottleneck point.

Inspired by CloudWatcher and QoSE, Duo adopts distributed traffic classifiers so that we address the latency problem that can be caused by the centralized traffic classification process. In addition, Bohatei motivates us to design the NFV-based traffic classifier, which helps efficiently distribute the classifiers over the data plane.

3.3. Intrusion Detection System. Intrusion detection system (IDS), such as Snort [1] and Suricata [24], is a reactive security system that defends a target system against intrusions (or successful attacks) by detecting attack trials [2]. It detects an attack trial and reports that incident to the network administrator, and then he or she decides which action to take to eliminate the threat. IDS can employ different approaches that are classified as anomaly- and signature-based detection in general [25], and both of them require signature or anomalous behavior patterns of attacks in advance to detect attacks. However, it is not often the case that such signatures and anomalous characteristics of attacks are available before the attacks show up, and thus IDS sometimes fails in detecting attack trials.

Different from IDS, the recovery-based ITS takes a proactive approach to eliminate threats from attackers. Aiming to remove any intrusions in the target system by self-rejuvenating methods [16], the recovery-based ITS assures the safety of the systems.

3.4. Network Traffic Classification. Network traffic classification has been an important research topic in network security community [26]. In our work, network traffic classification is deemed a process to seek a client's reputation in terms of security. In this context, our traffic classification is analogous to Peng and others' work [20] in that their history-based IP filtering admits incoming packets if their senders have already accessed a network's edge router normally and their IP addresses have been registered in a history-based IP database. However, if an adversarial user launches attacks after deceiving the history-based IP database by sending reconnaissance packets, then it is hard for history-based IP filtering to block attack packets. Our work, on the other hand, takes a conservative approach that inspects every incoming packet to reinforce the security performance.

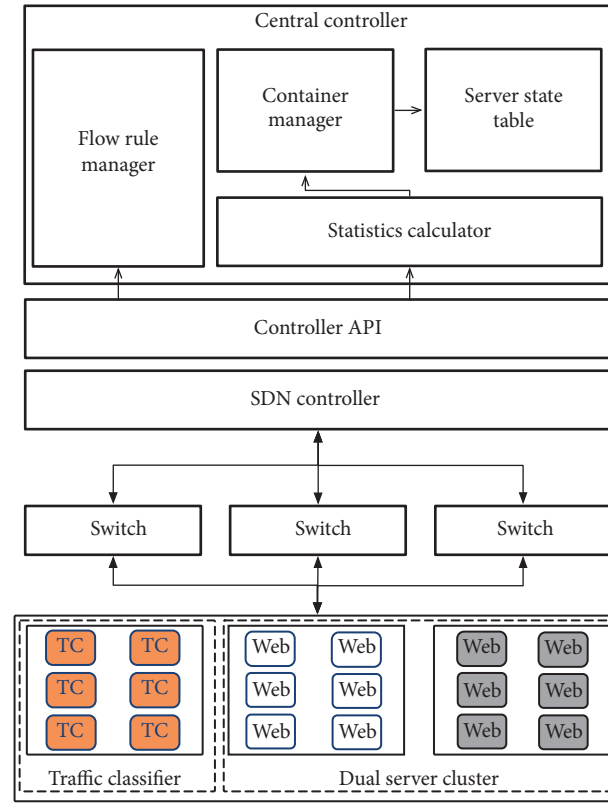


Figure 1: The architecture of Duo.

4. System Design

Duo is a new network intrusion tolerant system (ITS) that consolidates the security features of ITS with SDN. Since the main goal of Duo is to minimize exposure time of ITS by treating network requests differently, we employ SDN to control network requests flexibly in realizing Duo.

4.1. Architecture Overview. In Figure 1, we illustrate the architecture of Duo which is composed of three main components as follows: (i) central controller, (ii) dual server cluster, and (iii) traffic classifier.

Central controller manages the dual server cluster and SDN flow rules. To this end, the central controller has four modules as follows:

(i) *Statistics calculator* collects statistics on how many network flows in each type (i.e., suspicious and benign) are getting in to the system.

(ii) *Container manager* computes the optimal size of the dual server cluster and the proper number of traffic classifiers, using the statistics information provided by the statistics calculator. This module also creates container servers and traffic classifiers.

(iii) *Server state table* keeps track of servers' state change that follows the server lifecycle model.

(iv) *Flow rule manager* creates and installs a new flow rule if unknown network flow accesses the system.

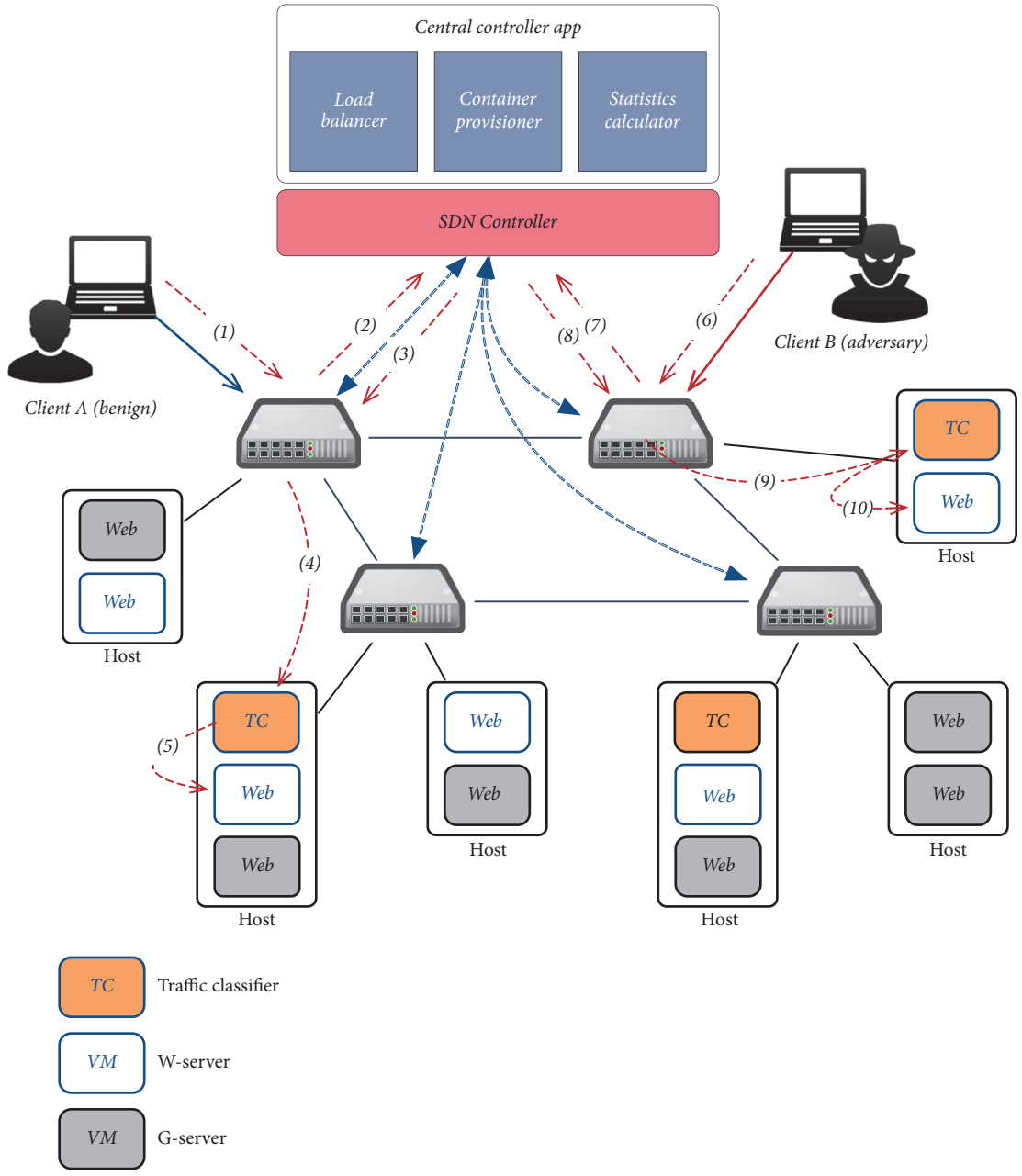


Figure 2: The overall working scenario before an attack is detected.

Moreover, this module updates already installed flow rules if benign network flows turn out to be malicious.

Dual server cluster is two groups of the servers, White and Gray clusters, and the servers in both clusters have the same functionalities but the exposure time. The two clusters are adaptively scaled as the central controller computes the appropriate size of each cluster in a periodic fashion.

Traffic classifier (TC) is a software appliance that determines whether network traffic is benign or not. On top of the NFV technology, Snort [1], an open source NIDS, is installed on multiple containers, which are distributed over the data plane. So, whenever a network flow enters the system, it should be inspected by one of the traffic classifiers.

4.2. Overall Working Scenarios. Now, we present the overall workflow of Duo by taking example scenarios as shown in Figure 2. When (1) an HTTP request arrives at Duo, the data plane searches for a flow rule that instructs how to handle the request. If the data plane cannot find any matching rules for the request, (2) it asks the control plane (or, more specifically, the central controller) what to do with the request, sending a `packet_in` message. Then, (3) the central controller installs a flow rule on the data plane, which sends the request to the TC

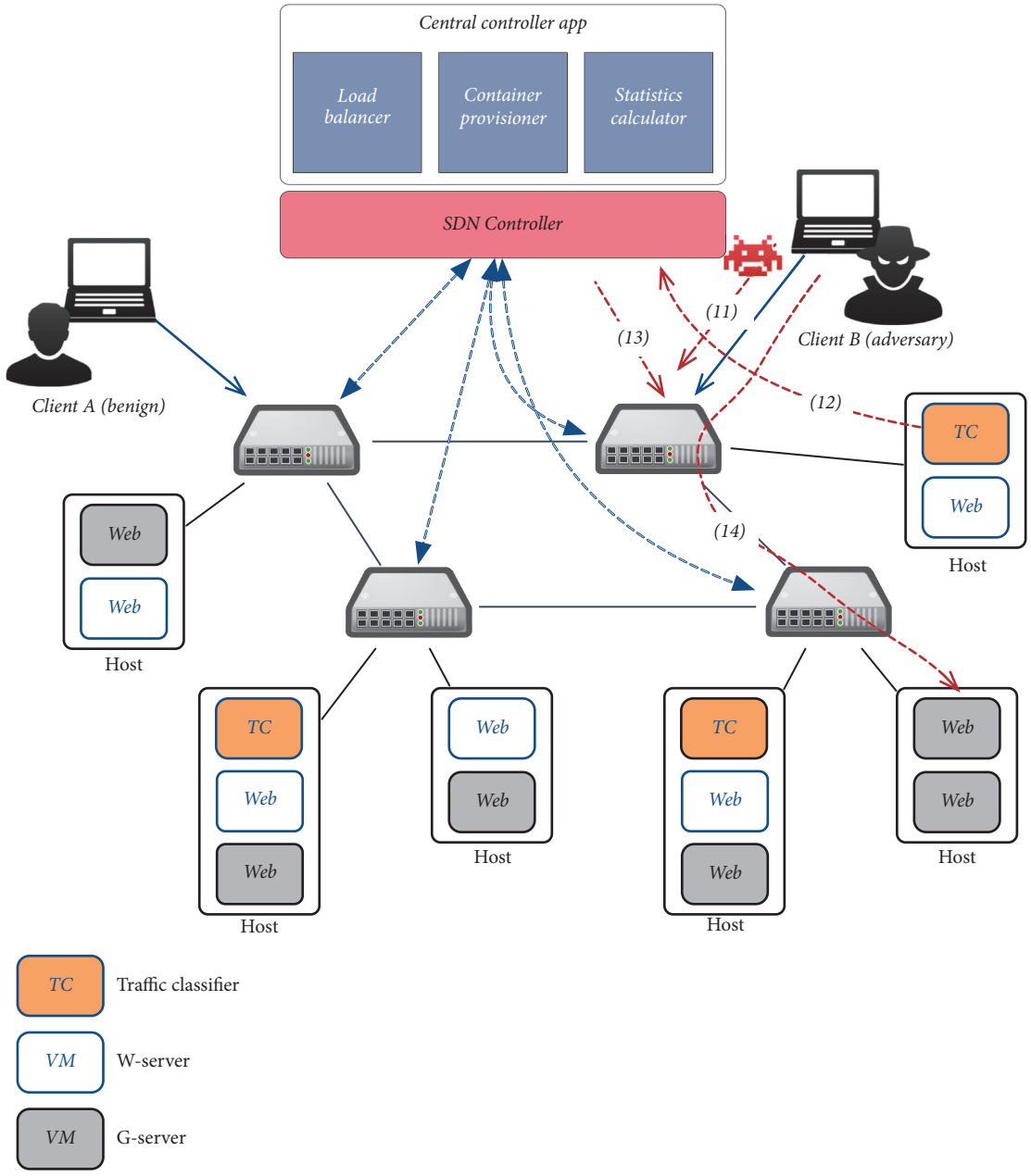


Figure 3: Working scenario when an attack trial is detected.

using a `flow_mod` message. (4) The TC inspects the request and decides the type of the request between two types: benign and suspicious. Suppose that (5) the request is determined as benign by the traffic classifier, and then this request is sent to a White server.

On the other hand, suppose that an attacker wants to access a server. In this case, Duo handles his packets as usual (i.e., from (6) to (10) in Figure 2). However, at some time point, as shown in Figure 3, (11) he will start an attack, and this trial will be recognized by the traffic classifier. Then, (12) the accident is reported to the central controller immediately, and (13) the previously installed flow rules for the attacker will be removed from the data plane. The central controller will install a new flow rule that describes (14) to forward the attacker's packets to a Gray server afterward.

4.3. Why ITS Classifies Network Flows. The key idea of Duo is to have different recovery schedules for the components (i.e., servers). Basically, Duo is a recovery-based ITS that recovers the servers to their pristine state in a periodic manner. However, unlike existing ITS studies, Duo does not employ a uniform recovery schedule. Rather, it classifies traffic into two types and gives the servers different exposure times depending on the type of traffic to which the servers are assigned.

Then, why do we bother to classify network flows even in designing the recovery-based ITS? The answer is that every flow does not leave the same footprint when it is served by the server. In other words, most of the users are benign and want to receive services reliably, but some adversarial users seek to make security breaches by infecting the server. Thus, if the server served illegitimate or suspicious flows, it is highly probable that the server would be polluted while serving them. On the other hand, if the flows are benign, possibly the server remains uncontaminated.

Under this premise, we divide the servers into two groups: White and Gray cluster. The two clusters have a different recovery schedule because they are in charge of serving the two different types of network flows: benign and suspicious flow. Benign flows are served by the servers in the White cluster; hence the servers do not have to recover frequently, which helps save resources. Suspicious flows, on the contrary, are served by the servers in the Gray cluster, which will recover more often than those in the White cluster. This is because the servers in the Gray cluster are believed to have pollutants during their operation.

4.4. How Duo Fortifies Security of ITS. The goal of Duo is to reduce the exposure time of a recovery-based ITS. Duo has a separate group of servers (i.e., Gray cluster) that are in charge of serving suspicious network traffic, whereas the rest of the servers (i.e., White servers) will handle benign network traffic. Since the Gray servers are more likely to be exposed to threats than the White servers, we give the Gray servers much shorter exposure time than the usual exposure time, which is assigned to the White servers. Intuitively, as we have assigned the shorter exposure time to a part of the servers, the average exposure time over the system becomes much shorter than when all the servers have the usual exposure time.

To achieve the goal, we addressed three main issues as follows:

(i) Duo employs the dual server cluster that constitutes two types of servers, White and Gray clusters, that are separated by their exposure time. The Gray servers are given shorter exposure time than that of the White servers because they will handle suspicious traffic.

(ii) Duo scales the dual server cluster adaptively, to cope with the change in volume of network traffic. The scaling process is operated by an ILP-based algorithm, which finds the optimal size of each server cluster within budget in terms of the container servers available.

(iii) To classify network traffic into suspicious and benign types, Duo employs traffic classifiers that are implemented on top of the NFV technology. To minimize overhead that can be caused by traffic classification, multiple traffic classifiers are distributed over the network.

4.4.1. Dual Server Cluster. The dual server cluster is two server clusters that consist of containerized servers. Duo distributes the container servers between the dual server cluster according to their exposure time. If a server is assigned to the White cluster, then its exposure time is set to E_{Long}. On the other hand, if the server is allocated to the Gray cluster, then the server's exposure time becomes E_{Short}. The difference between E_{Long} and E_{Short} is time duration for the server to be online; E_{Short} is defined much shorter than E_{Long} because the servers in Gray cluster serve suspicious requests that are more likely to be malicious.

By letting exposure time of a group of container servers much shorter than that of the other, we are able to reduce the overall exposure time of the entire system. Existing ITS architectures assign all the VM servers uniform exposure time without addressing how they select exposure time [8, 11, 12, 16, 27, 28]. However, we argue that the overall exposure time can be reduced without using additional resources if we assign each server unidentical exposure time in accordance with the type of clients that the server will handle.

To show how exposure time of Duo can be reduced, we define $\overline{E}$, *systemwide exposure time,* which represents the mean exposure time of the running servers of ITS. Unlike existing ITS architectures employing uniform exposure time, we employ two different exposure times, and thus it is not possible to perform a direct comparison of exposure time between Duo and other architectures. Hence, we compare the two systemwide exposure times, $\overline{E}_{\text{Duo}}$ for Duo and $\overline{E}_{\text{Uni}}$ for ITS with uniform exposure time:

$$\overline{E}_{\text{Duo}} = \frac{E_{\text{Long}} \cdot W + E_{\text{Short}} \cdot G}{N}, \tag{1}$$

$$\overline{E}_{\text{Uni}} = E_{\text{Uni}}, \tag{2}$$

$$E_{\text{Short}} < E_{\text{Long}} = E_{\text{Uni}}. \tag{3}$$

We define $\overline{E}_{\text{Duo}}$ as shown in (1), where W and G denote the number of each type of the running servers, respectively, and N is the sum of W and G (i.e., the total number of the running servers). Note that, in case of an ITS architecture with uniform exposure time, $\overline{E}_{\text{Uni}}$ is equal to its server's uniform exposure time, E_{Uni}, as shown in (2). Here, if we leave E_{Long} the same as E_{Uni} and assign E_{Short} to the Gray cluster, which is much shorter than E_{Uni}; then we conclude that it holds that $\overline{E}_{\text{Duo}} < \overline{E}_{\text{Uni}}$ as long as (3) is satisfied.

4.4.2. Resource Management. One pivotal role the central controller plays is to manage resources, or the servers. More specifically, the central controller recovers the servers following their lifecycle model and provisions the dual server cluster for additional servers when required.

Server Lifecycle Model. Before giving a detailed explanation of the lifecycle model, we need to discuss the server state table first. The central controller has the server state table, where state information of running servers is stored. A server's state is summarized in seven fields as described in Table 2, and the central controller keeps track of the server's state change from creation to destruction. The entries in the table are used for scheduling server recovery and making flow rules to refer to a location of servers.

Table 2: An entry of the server state table.

Server id	Type	No. of recv packets	No. of sent packets	Creation time	State	Location

When the central controller decides to create a server, it also generates a table entry for that server. Initially, except the number of the received and sent packets, all the fields are filled with values of a unique id, the type (i.e., White or Gray), the creation time, the state (i.e., initial creation), and the location (e.g., physical host id), respectively.

Now, we present the lifecycle model in detail as illustrated in Figure 4. When a server is created, its state becomes *creation*, and the timestamp for this event is recorded in the state table entry. With this timestamp, we will determine whether a server should be recovered or not. After created, the server soon receives the first packet, and its state transitions to *running*, which means it is ready to handle packets. This state transition event is also recorded in the state table entry.

When the server only has the grace time left within its operational time, then the server goes into *grace period* state. Here, the grace time denotes a time window during which the server does not accept new requests and processes the remaining requests in its queue. Giving the grace time to the server, we prevent users from experiencing a sudden interruption to the service provision. Finally, the server's state will be changed from *grace period* to *destruction* after the grace time expires, or all the remaining requests are served even if the grace time is not expired. At this state, the server will be stopped and destroyed.

In SCIT, the servers' lifecycle model is also proposed, but their lifecycle model differs from ours in that our containerized servers do not need to remain ready for a long time because they can be instantly created and become ready [7].

Server Provisioning. Another role the central controller plays is to increase or decrease the size of server clusters dynamically. If the volume of traffic increases, Duo creates and puts new servers into the clusters as far as computing resources allow. Although defending against a denial of service attack is beyond the scope of this work, this operation ensures availability to provide users with services in a timely way. On the other hand, to save resources, Duo decreases the size of server clusters when the central controller estimates that the number of running servers is too large for the volume of current traffic.

When a server is going to recover, another server should replace the server as long as the volume of traffic remains flat. Similarly, in case of an increase in the volume of traffic, additional servers should be supplied promptly to cope with the situation. In Duo, to satisfy this requirement and to achieve the main goal of reducing the systemwide exposure time, we take an optimization approach to the resource management problem. To be specific, we formulate the problem as an integer linear program (ILP) that is aimed at guaranteeing availability with the systemwide exposure time minimized.

In (4), we present the ILP formula for the resource management problem with the objective function to minimize the sum of the running servers' exposure time. The

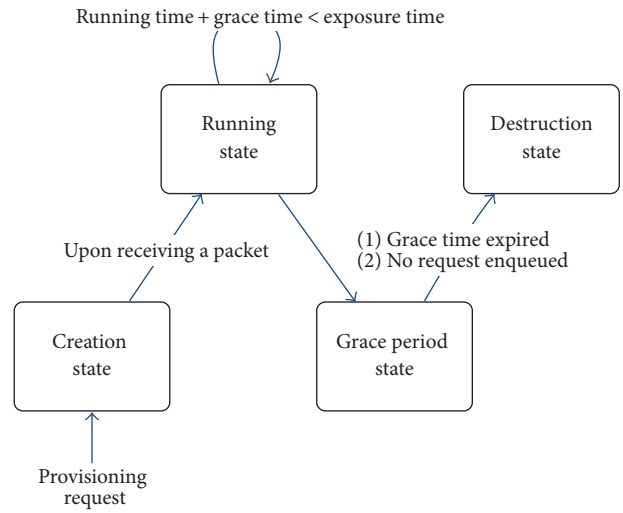


Figure 4: The lifecycle of a containerized server.

objective function is dependent on the variables x and y, which are the number of White and Gray servers in running state, respectively. K denotes a server's capacity to serve the requests, which means the maximum number of requests that a server can process for one second (rps). In addition, we define the volume of benign and suspicious traffic as BT and ST in rps, respectively:

$$\text{minimize} \quad \sum^{x} E_{\text{Long}} + \sum^{y} E_{\text{Short}} \tag{4}$$

subject to the following constraints:

$$x \geq \frac{\text{BT}}{K} \tag{5}$$

$$y \geq \frac{\text{ST}}{K} \tag{6}$$

$$x + y \leq \text{MAX} - T, \tag{7}$$

where

K is the capacity of a server (rps),

BT is the volume of benign traffic (rps),

ST is the volume of suspicious traffic (rps),

T is the number of traffic classifiers,

MAX is the max number of servers a system can make.

In order to handle requests without much delay, the system expects to have x White servers and y Gray servers. However, the expected total number of servers, $x + y$, should not exceed MAX $- T$, assuming that the system can have total

```
(1) def calcRank (switch, tc):
(2)    p = .5
(3)    state = tc.state
(4)    busy = tc.busyness
(5)    near = relativeDistance (switch, tc)
(6)    return state * (p * near + (1−p) * busy)
(7)
(8) def selectTC (switch):
(9)    ranks = list ()
(10)   for tc in trafficClassifiers:
(11)      rank = calcRank (switch, tc)
(12)      ranks.append (rank)
(13)
(14)   ranks = sorted (ranks, descendingOrder=True)
(15)   return ranks [0]
```

Algorithm 1: Traffic classifier selection algorithm.

MAX containers, out of which T containers are used as traffic classifiers. Satisfying these constraints, we periodically find the optimal values for the variables x and y.

4.4.3. Network Flow Management. Duo requires all new network flows to go through a TC before flow rules for the flows are activated. This approach, however, produces two challenging issues related to security and performance. First, an attacker can try to compromise the TC for the purpose of forwarding his malicious packets to the White cluster. Second, because the central controller checks the type of the flow and then installs a new rule for the flow, users may experience a long response delay if the network classification process takes much time to give a result.

We elaborate on two possible scenarios that an attacker prevents the TC from achieving its goal. First, we can imagine that the attacker compromises the TC in order to let the central controller make inappropriate flow rules. For example, if the attacker succeeds in taking control over the TC, he may make the classifier tag his packets as benign. Then, his packets will be forwarded to the White cluster, and he can get more time to attack the entire system afterward. The other scenario is that an attacker can launch resource exhaustion attacks (e.g., DoS attacks) to overload the TC with new network flows. In this case, since the TC is the entry point of the system, the throughput of the entire system can be dropped considerably.

Distributed Traffic Classifier. In order to address these issues, we (i) make the traffic classifiers intrusion tolerant and (ii) distribute them across the dual server cluster. Like the web servers, the TCs are also containerized and recovered to their clean state, which are basic intrusion tolerance defense operations that we believe can protect the TCs against attacks. Distributed TCs, on the other hand, are more related to availability. For instance, if the system has a single TC, then it cannot continue to provide users with services as soon as the TC fails. Therefore, we deploy multiple instances of the TC, as described in the previous section. However, we limit the number of TCs because they perform a single operation (i.e., classification).

To realize the distributed traffic classifiers, Duo deploys the TCs as a virtual network function (VNF) and distributes them across the dual server cluster. Recently, combined with SDN, the network function virtualization (NFV) technology fosters better security by enabling network functions to be virtualized. For example, without using a middlebox, we are able to install software IDS, such as Snort [1] or Suricata [24], on a VM and to initiate SDN flow rules in order for network traffic to make a detour through the VM. Then, network traffic will be inspected or monitored by the VM, more specifically the virtualized IDS.

Traffic Classifier Selection. If a packet arrives an edge switch of the system, and there does not exist a matching flow rule to the packet, then it is forwarded to a TC to be inspected. However, because there are distributed TCs, when we select a TC to classify the packet, we need to consider the routing overhead and the state of a TC. For example, Since the containerized TCs are distributed over the dual server cluster, there are many paths from the edge switch to a TC, and some of the paths may not be optimal. In addition, a TC can be busy when a large number of new packets are flocked to the classifier. Therefore, the central controller needs to make an efficient routing path for the new packets.

Therefore, we propose a TC selection algorithm as presented in Algorithm 1, considering the factors as follows:

(i) nearness: the relative distance from the edge switch to a TC,

(ii) busyness: the degree to which a TC is relatively busy,

(iii) state: the state of a TC, whether or not a TC is going to recover soon.

When we design the TC selection algorithm, we first consider how many nodes exist between the edge switch and a TC. Basically, the shorter the path from the switch to a TC is, the less the time it takes for a packet to be delivered. Next, the busyness of a TC is as important as the shortest path because a

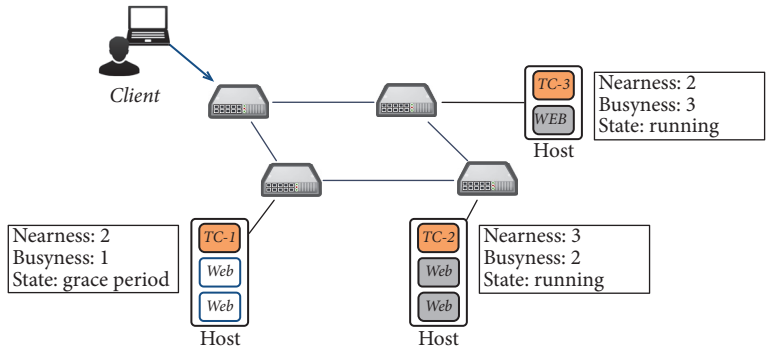


Figure 5: Example scenario of the traffic classifier selection algorithm.

busy TC may impose waiting time on the traffic classification. Last, the state of a TC is also a decision factor. Even if the path is short and a TC is idle, the TC is not available if it is about to recover soon. In other words, a TC in the grace period state cannot be selected:

$$\text{rank} = \text{state} * \left(p * \text{nearness} + (1 - p) * \text{busyness}\right). \quad (8)$$

In order to select the most appropriate TC, the algorithm first computes ranking values for each TC, which is presented in (8). There are three variables: state, nearness, and busyness. The variable *state* can have a binary value meaning whether or not a TC is in the running state. The variable *nearness* is defined as the distance between a TC and a client, and the variable *busyness* denotes how much a TC is busy. The last two variables have relative values. For example, as illustrated in Figure 5, if we assume the TC-3 is the busiest among the 3 TCs, then its busyness value becomes 3. In addition, since the hop count between the client and the TC-2 is the largest, the nearness of TC-2 becomes 3. Finally, the importance of the two variables is regulated by the regulation factor p, which lies between 0 and 1. For example, setting p to 0.5 indicates that we treat the importance of the variables equally.

4.4.4. Central Controller Protection. In Duo, the central controller is an intelligent part that is in charge of controlling server recovery and network flows. These operations are the main mechanisms for Duo to protect a victim system, and thus failure of the central controller can lead to failure of the victim system. However, since the central controller is an SDN application, it is moved to the control plane of SDN, and, as a result, its security is more fortified. Specifically, the central controller is isolated from the outside and thus protected from external threats. This is because communication between the control and data plane in SDN is performed in the OpenFlow protocol [29], and the data plane does not have the capability to affect the control plane. In addition, the central controller is deployed in a distributed fashion along with the SDN controller, and so availability of the central controller is also increased.

5. Performance Evaluation

5.1. Evaluation Environment. In order to evaluate the performance of the proposed system, we have implemented the prototype of Duo in the mininet environment [30]. Mininet allows building SDN prototypes easily, and it is widely used to develop and evaluate OpenFlow applications, providing a realistic network setting. In addition, we use ONOS [31] as the SDN controller, and we build the central controller on top of ONOS using its APIs.

As illustrated in Figure 6, we compose the evaluation topology, which consists of four switches, two clients, and lots of virtual servers including traffic classifiers. To build the servers and the traffic classifiers, we use mininet hosts that are container-like virtual machines. We create a web server that is running on each server, and a copy of Snort is installed on each traffic classifier. Inspired by FlowTags [32], we modify Snort to tag packets if it finds the packets suspicious. In particular, for a realistic performance evaluation, we used real web server logs collected for one year, and the web server is maintained by a software vendor to advertise their products. Since the web server consists of WordPress to publish the contents, we can observe considerable attack trials related to WordPress. Finally, we replay the web logs using Apache JMeter [33].

5.2. Resource Management. In Figure 7, we present the status change of the servers and how they increase or decrease in number. Specifically, there are three boxes in red, which indicate the number of (1) the running servers and (2) additionally required servers to process incoming traffic. If we see the box at the top, the two numbers are the same, which means the volume of traffic remains flat. On the contrary, in the box at the middle, currently there are 4 Gray servers running, and Duo determines that there should be 21 more servers to deal with current traffic. However, due to resource limitation, Duo can add 16 more servers only. Eventually, as shown in the last box, 16 more servers are added, and thus there are 20 Gray servers running.

5.3. Reduced Exposure Time. In this experiment, we compare two ITS architectures and discuss how they are different in

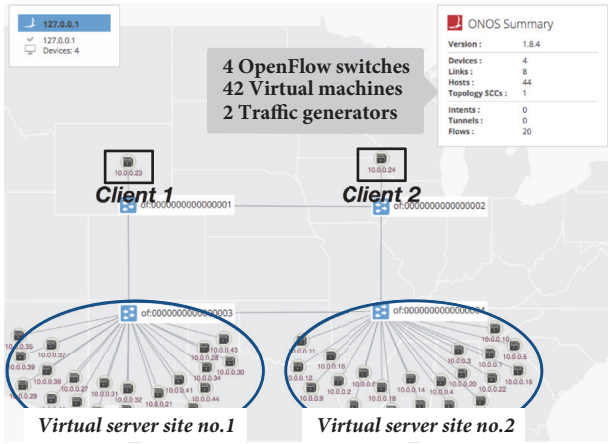


Figure 6: Evaluation environment settings implemented with ONOS.

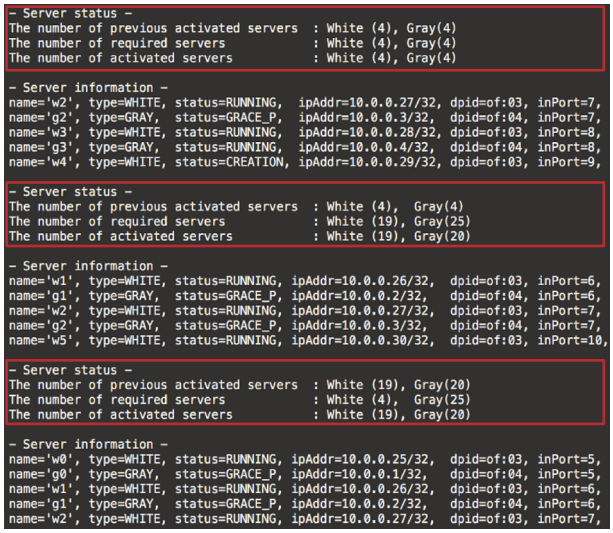


Figure 7: Server status change according to the lifecycle model and server provisioning process.

terms of response time and exposure time. For this, we built a baseline system that does not classify traffic and employs servers with the same exposure time, E_{Uni}, of 60 s, and then we compare the baseline system and Duo. In case of Duo, we use 60 s and 30 s as E_{White} and E_{Gray}, respectively.

In Figure 8, we show the average exposure time of the baseline system and Duo. According to the volume of two types of traffic, the White and Gray servers are increased and decreased in number. For example, at time point 6, there is more suspicious traffic than benign one, and hence Duo added more Gray servers, which results in decreasing the average exposure time. On the other hand, at time point 9, more benign traffic enters Duo, so White servers are created to handle those traffic.

The average exposure time of Duo, $\overline{E}_{\text{Duo}}$, is always shorter than that of the baseline system, E_{Uni}. Since we set E_{Uni} and E_{White} the same, and E_{Gray} is shorter than them, E_{White} acts as

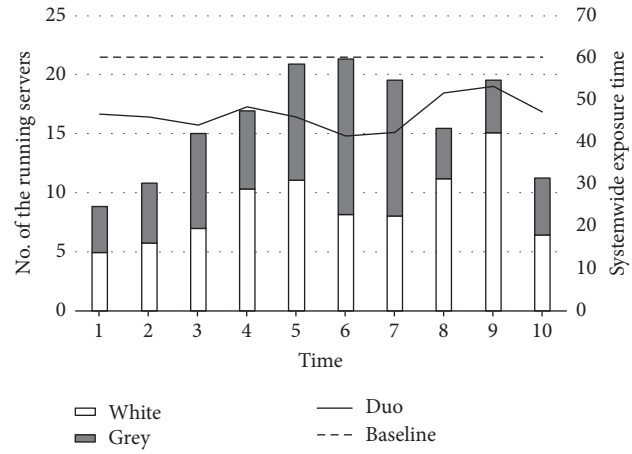


Figure 8: Average exposure time and change in the size of dual server cluster.

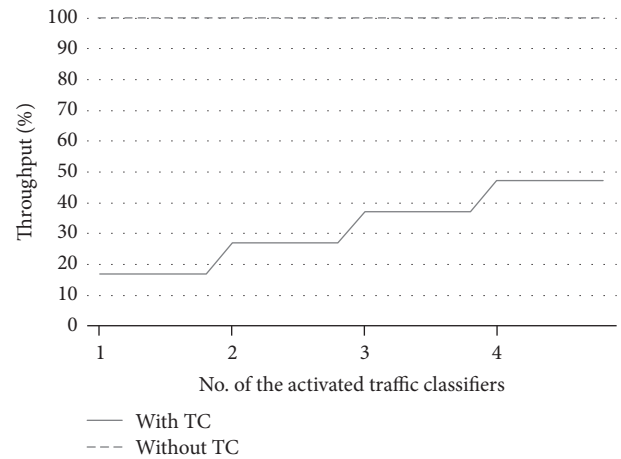


Figure 9: Change in throughput according to the number of the activated traffic classifiers.

the upper bound of the average exposure time as defined in (1).

5.4. Traffic Classification Overhead. Another concern that needs to be addressed is how Duo can reduce the traffic classification overhead. Since Duo requires incoming network requests to be inspected by a traffic classifier, throughput of the system can be dropped if an insufficient number of traffic classifiers are deployed. To resolve this problem, Duo activates or inactivates traffic classifiers on the fly according to change in the volume of network traffic.

In Figure 9, we illustrate how the additionally activated traffic classifiers can increase throughput of the system. To conduct this experiment, we built a test bed that is composed of multiple container servers on physical machines and SDN-compatible switches, and the test bed has a network topology described in Figure 5. As shown in Figure 9, when compared to the system without traffic classification process, Duo shows about 20% of throughput when a single TC is employed. However, throughput of the system is gradually increased

as extra TCs are activated, and in our settings, the result implies that Duo requires about eight TCs to minimize the classification overhead.

6. Discussion

We now discuss some limitations in our work. First, Duo employs the existing NIDS as the traffic classifier, which is sometimes unable to classify network traffic correctly. For example, if there is a zero-day attack, whose signature is not disclosed yet, then our traffic classifier might tag traffic that the attack generates as benign. However, the exposure time of the White servers could be selectively tuned to be much shorter than usual in order to eliminate such an attack from the servers even if the attack succeeds in infecting them. Thus, Duo can focus more on strengthening the security performance.

Second, in the current design, every network flow is inspected by the TC, which introduces new latency. However, attackers may send benign packets first and attack packets next. So, if we can model the time when attackers start to send suspicious packets, then we are able to send the suspicious packets directly to the Gray servers after inspecting packets only during the period of the modeled time. Extending Duo to reduce the time that network traffic should pass through a traffic classifier is future work.

7. Conclusion

In this paper, we propose Duo, an ITS incorporated in SDN, which is aimed at reducing exposure time by classifying network traffic into two types: suspicious and benign. According to the classification result, suspicious traffic is forwarded to Gray servers and benign one to White servers. Since the Gray servers are more likely to handle malicious traffic, we assign them much shorter exposure time than that of the White servers, which can reduce the average exposure time. To achieve this, we should address two key challenges; one is classification bottleneck problem and the other is server management considering the volume of each type of traffic. To address the first challenge, we employ SND and NFV technologies that enable centralized and adaptive network flow management. With the help of SDN/NFV, we perform the traffic classification in a distributed way. To resolve the second challenge, we formulate the server management problem taking an ILP approach. Our performance evaluation shows that Duo reduces the average exposure time even if the volume of traffic changes. To the best of our knowledge, Duo is the first work that incorporates ITS into SDN.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors thank Jaehyun Nam for his comments on initial design and his help in testing. This work was supported by Institute for Information & Communications Technology Promotion (IITP), a grant funded by Korea government (MSIP) (no. 2016-0-00078, Cloud Based Security Intelligence Technology Development for the Customized Security Service Provisioning).

References

[1] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration LISA '99*, pp. 229–238, USENIX Association, California, Calif, USA, 1999.

[2] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: methods, systems and tools," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 303–336, 2014.

[3] A. Saidane, V. Nicomette, and Y. Deswarte, "The design of a generic intrusion-tolerant architecture for web servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 45–58, 2009.

[4] P. Verssimo, N. Neves, and M. Correia, "Intrusion-tolerant architectures: concepts and design," in *Architecting Dependable Systems*, R. D. Lemos, C. Gacek, and A. Romanovsky, Eds., vol. 2677 of *Lecture Notes in Computer Science*, pp. 3–36, Springer Berlin Heidelberg, Berlin, Germany, 2003.

[5] Q. L. Nguyen and A. Sood, "A comparison of intrusion-tolerant system architectures," *IEEE Security & Privacy*, vol. 9, no. 4, pp. 24–31, 2011.

[6] D. Wang, B. B. Madan, and K. S. Trivedi, "Security analysis of sitar intrusion tolerance system," in *Proceedings of the 2003 ACM Workshop on Survivable and Self-regenerative Systems: In Association with 10th ACM Conference on Computer and Communications Security, SSRS '03*, pp. 23–32, ACM, New York, NY, USA, October 2003.

[7] A. K. Bangalore and A. K. Sood, "Securing web servers using self cleansing intrusion tolerance (SCIT)," in *Proceedings of the 2009 2nd International Conference on Dependability, DEPEND 2009*, pp. 60–65, Greece, June 2009.

[8] S. Heo, S. Lee, B. Jang, and H. Yoon, "Designing and implementing a diversity policy for intrusion-tolerant systems," *IEICE Transaction on Information and Systems*, vol. 94-D, no. 4, pp. 1–12, 2016.

[9] B. B. Madan, K. Goševa-Popstojanova, K. Vaidyanathan, and K. S. Trivedi, "A method for modeling and quantifying the security attributes of intrusion tolerant systems," *Performance Evaluation*, vol. 56, no. 1–4, pp. 167–186, 2004.

[10] Q. Nguyen and A. Sood, "Quantitative approach to tuning of a time-based intrusion-tolerant system architecture," in *Proceedings of the 3rd Workshop Recent Advances on Intrusion-Tolerant Systems*, pp. 132–139, 2009.

[11] B. Jang, S. Doo, S. Lee, and H. Yoon, "Hybrid recovery-based intrusion tolerant system for practical cyber-defense," *IEICE Transaction on Information and Systems*, vol. E99D, no. 4, pp. 1081–1091, 2016.

[12] J. Lim, Y. Kim, D. Koo, S. Lee, S. Doo, and H. Yoon, "A novel Adaptive Cluster Transformation (ACT)-based intrusion tolerant architecture for hybrid information technology," *The Journal of Supercomputing*, vol. 66, no. 2, pp. 918–935, 2013.

[13] J. Lim, S. Doo, and H. Yoon, "The design of a robust intrusion tolerance system through advanced adaptive cluster transformation and vulnerability-based VM selection," in *Proceedings of the 2013 IEEE Military Communications Conference, MILCOM '13*, pp. 1422–1428, USA, November 2013.

[14] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[15] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[16] Q. L. Nguyen and A. Sood, "Designing SCIT architecture pattern in a cloud-based environment," in *Proceedings of the Dependable Systems and Networks Workshops (DSN-W), IEEE/IFIP 41st International Conference*, pp. 123–128, IEEE, Hong Kong, June 2011.

[17] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks or: How to provide security monitoring as a service in clouds?" in *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP '12)*, pp. 1–6, 2012.

[18] T. Park, Y. Kim, J. Park, H. Suh, B. Hong, and S. Shin, "Qose: quality of security a network security framework with distributed nfv," in *Proceedings of the IEEE International Conference on Communications, ICC '16*, pp. 1–6, May 2016.

[19] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic ddos defense," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pp. 817–832, USENIX Association, Washington, Wash, USA, 2015.

[20] T. Peng, C. Leckie, and K. Ramamohanarao, "Protection from distributed denial of service attacks using history-based IP filtering," in *Proceedings of the 2003 International Conference on Communications (ICC '03)*, vol. 1, pp. 482–486, usa, May 2003.

[21] K. Goseva-Popstojanova, F. Wang, R. Wang et al., "Characterizing intrusion tolerant systems using a state transition model," in *Proceedings of the DARPA Information Survivability Conference and Exposition II, DISCEX '01*, vol. 2, pp. 211–221, 2001.

[22] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: state-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[23] S. Shin, H. Wang, and G. Gu, "A first step toward network security virtualization: from concept to prototype," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 10, pp. 2236–2249, 2015.

[24] The Open Information Security Foundation (OISF), Suricata, https://suricata-ids.org/about.

[25] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: a comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[26] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems SIGMETRICS '05*, pp. 50–60, ACM, New York, NY, USA, 2005.

[27] S. Heo, J. Lim, M. Lee, S. Lee, and H. Yoon, "A novel intrusion tolerant system based on adaptive recovery scheme (ARS)," in *IT Convergence and Security 2012*, K. J. Kim and K.-Y. Chung, Eds., vol. 215 of *Lecture Notes in Electrical Engineering*, pp. 71–78, Springer, Dordrecht, Netherlands, 2013.

[28] Y. Kim, J. Lim, S. Doo, and H. Yoon, "The design of adaptive intrusion tolerant system (ITS) based on historical data," in *Proceedings of the International Conference for Internet Technology and Secured Transactions*, pp. 662–667, December 2012.

[29] Open Networking Foundation, OpenFlow, https://www.opennetworking.org/.

[30] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 19:6, 19:1 pages, ACM, New York, NY, USA, October 2010.

[31] P. Berde, M. Gerola, J. Hart et al., "ONOS: Towards an open, distributed SDN OS," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN '14*, pp. 1–6, ACM, New York, NY, USA, August 2014.

[32] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '13*, pp. 19–24, ACM, New York, NY, USA, August 2013.

[33] Apache JMeter, http://jmeter.apache.org.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 5650205, 13 pages
https://doi.org/10.1155/2018/5650205

Research Article

FAS: Using FPGA to Accelerate and Secure SDN Software Switches

Wenwen Fu, Tao Li, and Zhigang Sun

College of Computer, National University of Defense Technology, Changsha, Hunan 410073, China

Correspondence should be addressed to Wenwen Fu; fuwenwen94@163.com

Received 12 October 2017; Accepted 17 December 2017; Published 17 January 2018

Academic Editor: Chengchen Hu

Software-Defined Networking (SDN) promises the vision of more flexible and manageable networks but requires certain level of programmability in the data plane to accommodate different forwarding abstractions. SDN software switches running on commodity multicore platforms are programmable and are with low deployment cost. However, the performance of SDN software switches is not satisfactory due to the complex forwarding operations on packets. Moreover, this may hinder the performance of real-time security on software switch. In this paper, we analyze the forwarding procedure and identify the performance bottleneck of SDN software switches. An FPGA-based mechanism for accelerating and securing SDN switches, named FAS (FPGA-Accelerated SDN software switch), is proposed to take advantage of the reconfigurability and high-performance advantages of FPGA. FAS improves the performance as well as the capacity against malicious traffic attacks of SDN software switches by offloading some functional modules. We validate FAS on an FPGA-based network processing platform. Experiment results demonstrate that the forwarding rate of FAS can be 44% higher than the original SDN software switch. In addition, FAS provides new opportunity to enhance the security of SDN software switches by allowing the deployment of bump-in-the-wire security modules (such as packet detectors and filters) in FPGA.

1. Introduction

Software-Defined Networking (SDN) is a transforming networking design that simplifies network management and improves programmability of network [1]. Its basic attributes include the separation of control and data planes, logically centralized control of networks, and flexible and open interface to program underlying network infrastructure. In the architecture of SDN, the southbound interface is responsible for the interaction of network states between the control and data planes. Furthermore, it defines the forwarding abstraction of SDN data plane.

As SDN offers some hope for rapid prototyping and deployment, the data plane must provide mechanisms to deploy new network protocols, header formats, and functions, yet it still forwards traffic as fast as possible. OpenFlow is the most widely used standard SDN southbound interface which is a vendor-independent interface to switching elements [2, 3]. Most SDN switches are thus OpenFlow SDN Switches.

There are mainly two ways to implement the SDN data plane: hardware and software. Some SDN hardware switches, developed by the vendors of HP, NEC, and Arista, utilize customized ASIC switching chip. Although the ASIC way can get high forwarding performance, it can hardly be extended for new protocols and functions. Programmable hardware switches [4, 5] can provide the flexibility at the expense of large amount of hardware resources, such as expensive TCAMs for flow entries.

The SDN software switches are the most flexible to support new network services, new protocols, and new functions [6–9]. Unlike the TCAMs in hardware SDN switches, memory resource for accommodating flow rules is abundant in software SDN switches [10]. Thus, they are primary choices for SDN researchers in laboratories and have been widely deployed as first-hop virtual switches in data centers. However, a purely software-based approach can hardly satisfy the strict performance and line-speed security requirements of most modern networks.

With the continuous improvement of the capacity and the computing power of FPGA (Field Programmable Gate Array), we try to exploit benefits of FPGA on processing packets. It is noteworthy that Microsoft has built FPGA fabric attached to each server to accelerate large-scale data center services with customized function logic [11]. Due to its high performance with flexible reconfigurability, FPGA is also a good choice for accelerating and securing SDN software switches. We propose FAS (FPGA-Accelerated SDN software switch) to enable the offloading of time-consuming software functional modules and implementation of the real-time security modules in SDN switch processing path. The mechanism can address the performance shortage nicely while retaining the flexibility of software switches. In addition, it can also enhance the security property of software switches by deploying bump-in-the-wire security modules in FPGA. The contributions of this paper are summarized as follows.

(1) We make a comprehensive survey on current SDN switches in both academia and industry and classify the existing implementation models of SDN switches into different categories.

(2) We analyze the bottlenecks in SDN software switches on modern commodity multicore platform.

(3) We design FAS mechanism to offload functions in the forwarding path of SDN software switch, including packet buffer management, packet parsing, and some action executions for packets, to FPGA hardware.

(4) We implement the prototype of FAS on NetMagic-Pro, an FPGA-based network processing platform, and compare the performance with the original SDN software switches on commodity multicore platform.

The remainder of the paper is organized as follows. In Section 2, we review the evolvement of OpenFlow specification and introduce current SDN switches and their implementation models. Section 3 analyzes the overhead of OpenFlow forwarding in SDN software switches and points out the bottlenecks. In Section 4, we put forward a mechanism to offload software procedures of OpenFlow forwarding path to FPGA. Section 5 describes detailed design of the FAS mechanism implemented on an FPGA-based network processing platform (i.e., NetMagic-Pro). In Section 6, we give the performance comparison of FAS and the original SDN software switch. We summarize our work in Section 7.

2. Background and Related Work

In this section, we firstly make a brief introduction of the evolution of OpenFlow protocol, which challenges the design of SDN switches. Then we describe implementation models of SDN switches in both academia and industry.

2.1. Evolvement of OpenFlow. OpenFlow is the first and prevailing standard defined by Open Network Foundation (ONF) among all SDN southbound interfaces. It is wildly supported by many commercial switches, including HP, NEC, Arista, and Pica 8, and the list is still continually growing.

Since the first version (v1.0) distributed by ONF in December 2009, the specification of OpenFlow has been updated to version 1.5 in 2015 and has grown increasingly more complicated [3]. Although many features have been added to OpenFlow, the core concept of OpenFlow has not changed. OpenFlow switches process and forwards traffic on the basis of flows instead of individual packets. In the first version, the data plane abstraction is a single flow table of flow rules which could match packets on 12 header fields (e.g., MAC addresses, IP addresses, and TCP/UDP port numbers). In version 1.5, the OpenFlow switch has a pipeline of flow tables, where each flow rule has match fields (41 fields in the packet header), instructions (e.g., drop, flood, forward, or send the packet to the controller), a set of counters (to track the number of bytes and packets), a priority (to disambiguate between rules with overlapping patterns), timeouts (expiration time of flow rules), cookie (opaque data value chosen by the controller), and flags (to alter the way flow entries are managed). Upon receiving a packet, an OpenFlow switch identifies the highest-priority matching rule, performs the associated actions, and increments the counters.

With the evolvement of OpenFlow specification, OpenFlow has been extended with more capabilities and functions. As a result, the procedure of OpenFlow processing is getting more complicated and proliferating with no sign of stopping. It makes challenges to the implementation of SDN switches.

2.2. Implementation Models of SDN Switches. According to the OpenFlow specification, an SDN switch usually consists of four components: OpenFlow Channel (OFCh), OpenFlow Forwarding pipeline (OFFw), and physical port (Port). Besides, to accelerate multiple-tuple classification procedure in OFFw, a commonly used technique, Flow Cache (FCa), is introduced in SDN switches [12]. The descriptions of the four modules are as follows.

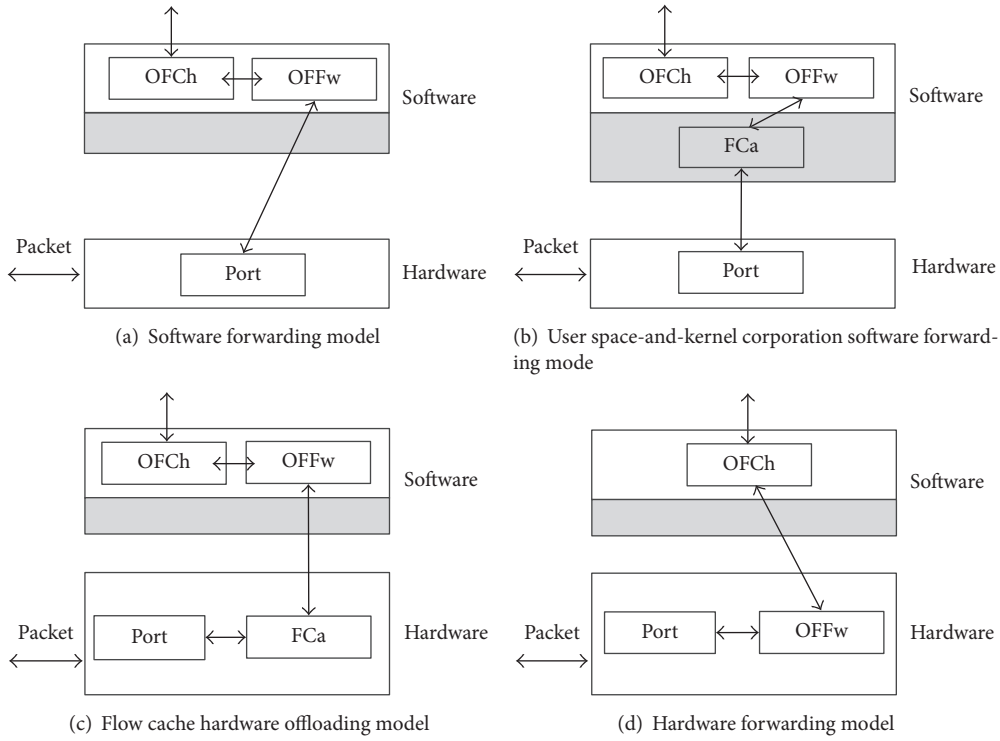
Port. It is the interface between the network and the switch and is responsible for packet receiving and sending.

FCa. It is the fast forwarding path for OFFw, which caches the entries recently matched in OFFw. Since network traffic has sufficient locality to provide high cache hit rate with relatively small cache size, FCa can accelerate the forwarding rate by bypassing OFFw.

OFFw. It maintains flow table of entries, in which each entry contains a set of packet fields to match and the corresponding actions to perform (e.g., forwarding, dropping, and modifying header).

OFCh. In the event when a switch does not find a match in OFFw, the packet is forwarded to the controller through OFCh. After deciding how to forward the new flow, the controller sends OpenFlow messages to the required switches. Then OFCh resolves those messages, generates flow rules, and installs them to the flow table. Besides, OFCh is also responsible for exchanging the states between the controller and the switch.

SDN switches have been implemented on different platforms (e.g., general CPUs, fixed function switch ASICs, reconfigurable hardware, NPUs, and FPGAs). The platforms can be divided into two categories: hardware-based switches



(a) Software forwarding model

(b) User space-and-kernel corporation software forwarding mode

(c) Flow cache hardware offloading model

(d) Hardware forwarding model

Figure 1: Current implementation models of SDN switches.

and software-based switches. Figure 1 shows different implementation models of the existing SDN switches. In the hardware-based switches, FCa or OFFw is implemented by hardware, such as ASICs and FPGAs. For example, Naous et al. implement an OpenFlow switch on Stanford's NetFPGA platform [13]. Pongrácz et al. devise an NP-based SDN switch to enhance the programmability in the data plane [14]. Some companies, such as Pica 8, supply commodity OpenFlow switches based on ASICs switch chips. These hardware switches are based on the Flow Cache hardware offloading model (in Figure 1(c)). RMT [4] and FM6000 [5] devise reconfigurable match tables in the OpenFlow pipeline, and they conform to the hardware forwarding model in Figure 1(d). The SDN hardware switches can provide sufficient forwarding capability, but they are costly and inflexible.

Recent improvement in processing power of multicores has given reason to revisit software switching. Due to its high flexibility and short development cycle, the SDN software switches are getting increased attention. Software switches commonly use commodity off-the-shelf (COST) PCs or servers equipped with multicores and multiple Network Interface Cards (NICs), running on general-purpose operation systems (such as Linux). The general-purpose OS provides a comfortable environment for research and development. OpenFlow reference switch (maintained by ONF) [6], OFSoftSwitch (maintained by CPqD) [7], and OpenFlow Click (maintained by Stanford) [8] are the typical SDN software switches referred to the software forwarding model (in Figure 1(a)). Open vSwitch [9] (namely, OVS, maintained by VMware) is an SDN software switch belonging to the user space-kernel cooperation model as shown in Figure 1(b). Table 1 gives the detailed information of above-mentioned SDN switching platforms.

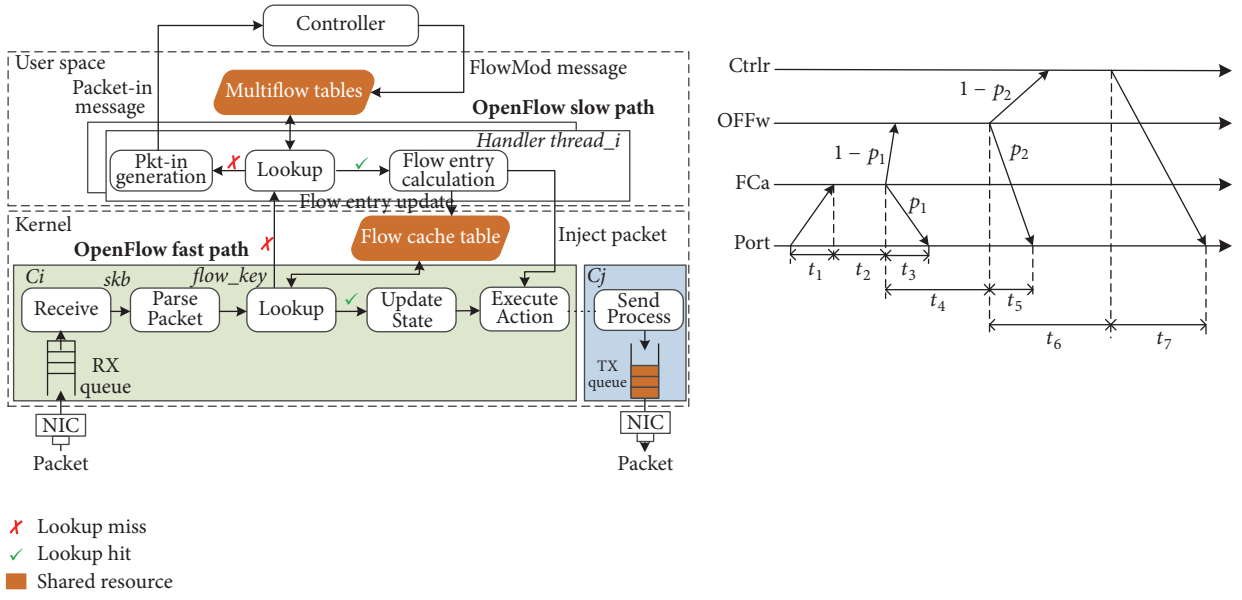
3. Problem Description and Analysis

3.1. OpenFlow Forwarding in SDN Software Switches. As the OVS is the most advanced and widely used SDN software switch, we focus on the user space-and-kernel cooperation (UKC) model as shown in Figure 1(b). Figure 2(a) illustrates the forwarding process of UKC model, while Figure 2(b) shows the timeline of it. For simplicity, we first discuss the procedure under multicore architecture with single-queue Network Interface Cards (NICs). The case with multiple-queue NICs will be discussed in Section 3.2. Without loss of generality, we assume that NET_RX and NET_TX of hardware interrupts in NIC i are both served by a fixed CPU core i. Cores i and j are denoted as Ci and Cj in Figure 2(a). If i equals j, packets are sent and received through the same NIC and processed by the same CPU core.

When a packet arrives at NIC i, this packet is attached to a descriptor in the NIC i's receiving (namely, RX) queue. The descriptor indicates the memory locations to store the incoming packets via Direct Memory Access (DMA) transfer. There are two data structures in the network stack of Linux Kernel. data_buff of 2 KB size is to hold the packet itself. The other data structure is sk_buff, which carries packet information metadata (e.g., pointer of packet data, MAC header, IP header, and packet states) used by TCP/IP protocol stack. The size of sk_buff is about 250 bytes. The sk_buff has a

Table 1: SDN switching platforms.

Names	Supporting OF version	Languages/hardware type	Application scenarios	Reference model
NetFPGA-based OpenFlow switch	1.0	FPGA	Research	Figure 1(c)
NP-based OpenFlow switch	1.0	Network processor	Research, enterprise	Figure 1(c)
Commodity OpenFlow switch	1.0	ASIC	Research, enterprise, data center	Figure 1(c)
RMT	All	ASIC	Research, enterprise	Figure 1(d)
FM6000	All	ASIC	Research, enterprise	Figure 1(d)
OpenFlow reference switch	All	C	Research	Figure 1(a)
OFSoftSwitch	After 1.3	C	Research	Figure 1(a)
OpenFlow Click	All	C++	Research	Figure 1(a)
Open vSwitch	All	C & kernel C	Research, virtualized data center	Figure 1(b)



(a) Forwarding process of UKC model

(b) Timeline of UKC forwarding

Figure 2: UKC implementation models of SDN software switch.

pointer to data_buff, and it makes up a Skb with the data_buff. After hardware interrupt of packet incoming is served, the Software Interrupts (SoftIRQs) are scheduled afterwards to accomplish the subsequent packet processing. The SoftIRQs are also bound to the specific processors.

In OVS, the handler function of SoftIRQ of *Ci* parses the packet by extracting all related match fields from Skb and stores them to flow_key.

The flow_key is used to look up the Flow Cache, which is shared among all processors in the kernel. If the Flow Cache contains the flow_key value, the Flow Cache will return instructions for processing the packet. Then it updates corresponding states and executes actions to the packet. If the packet is to be delivered by NIC *j*, it will be placed in the sending (namely, TX) queue of the NIC *j*. The sending process will be called in the function of NET_TX SoftIRQ of *Cj*.

If the lookup of Flow Cache is missed, the packet will be sent to the user space of Linux via the communication mechanism between the kernel and the user space, for example, netlink. Multiple handler threads in the user space will be created when the OpenFlow software switch is set up. These threads are responsible for processing mismatched packets coming from the kernel. The handler threads look up the shared multiflow tables. If the packets match with the tables, the handler threads will calculate for new Flow Cache entries and update them into the Flow Cache. The handler will also reinject the packet back to kernel for executing corresponding actions on the packet. If not, a Packet-in message will be generated and sent to the controller. The controller responds

with a FlowMod message to install corresponding flow rule for the packet to the multiflow tables.

Based on the analysis of the forwarding process, we evaluate the processing overhead in SDN software switches. We make some notations to define the time slots in the procedure. The time period from the packet arriving at the NIC to the lookup of the cache is denoted as t_1. The time of Flow Cache lookup is t_2. The time period from the start of matching Flow Cache to the end of sending out the packets is t_3. In the lookup, we denote the cache hit rate as p_1. The time period from the start to the end of looking up multiflow tables of user space is t_4. The time period from the beginning of matching multiflow tables to the end of sending out the packet is t_5. We denote the hit rate of multiflow tables as p_2. The time period from the start of triggering the controller to the end of installing the rule is t_6. At last, the time used for sending out the packet is t_7. Given these notations above, the evaluated total processing time T is formulated as follows:

$$T = t_1 + t_2 + p_1 t_3 + (1 - p_1)\, p_2\, (t_4 + t_5) + (1 - p_1)(1 - p_2)(t_6 + t_7). \quad (1)$$

Due to the high locality of network traffic and the proactive flow rule setup, the Flow Cache can get a high hit rate. As verified by Pfaff et al., the overall cache hit rate of Open vSwitch was 97.7% (i.e., the value of p_1) in a real commercial multitenant data center [15]. Thus, in most instances, the total process time T of a packet in SDN software switches can be approximately calculated as

$$T = t_1 + t_2 + t_3. \quad (2)$$

These three periods constitute the fast path of the OVS.

As depicted in Figure 2(a), the OpenFlow fast path consists of six functional modules: Receive, Parse Packet, Lookup, Update State, Execute Action, and Send Process. To understand the overhead for each functional module, we run Open vSwitch in a commodity PC as a software switch. We use the tool of Iperf [16] for generating the input traffic which is running over two 1 GbE NICs of the PC. Then we use Oprofile [17] to count CPU cycles for each function in fast path of Open vSwitch. We group all functions into the above six functional modules and the experimental results are shown in Figure 3. As we can observe, the Lookup module is the major bottleneck in the fast path. The other modules consume up to 44% of processing time in total.

3.2. Bottlenecks in SDN Software Switch. There are three main bottlenecks (i.e., packet I/O, software forwarding, and OpenFlow classification) in SDN software switch.

Packet I/O. Unlike CPU-intensive tasks, packet I/O is a critical step in software packet forwarding. Packet receiving and sending consume a large amount of CPU cycles. As pointed out in [18], the buffer allocation and release in packet I/O are the two major overheads, which represent up to 54% of total overhead.

Researchers have proposed several optimizing techniques such as Skb recycle queue, memory mapping, batch processing, affinity, and software prefetching to accelerate packet I/O. By combining some of above techniques, Intel develops a high-performance packet processing architecture on x86 platforms, named Data Plane Development Kit (DPDK) [19]. But the DPDK framework needs the support of specific CPU and NIC, which is not general. Rizzo designs a novel framework for fast packet I/O in general-purpose OS, named netmap [20]. However, netmap is originally run in the user space of OS. The performance drops when it is applied to kernel-based SDN software switches.

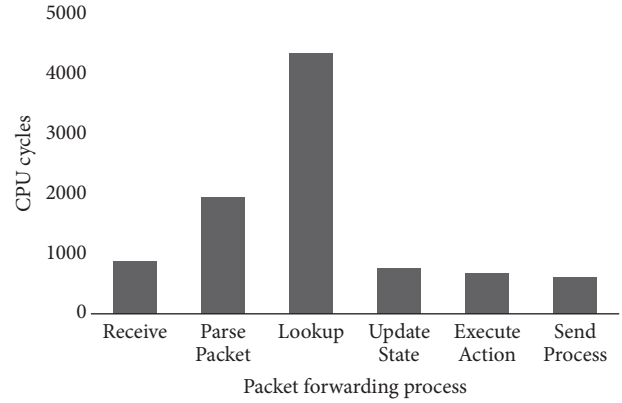


Figure 3: Packet forwarding process overhead breakdown. The processing for packets in Execute Action module only contains the forwarding operation.

Software Forwarding. A critical problem is contention for shared resources—caches, queues in NICs, and flow tables—in the case of multiple threads running concurrently to forward packets [21]. In order to improve the processing performance of SMP Linux, NIC-based core affinity is proposed to exploit the parallel packet processing capability of multicore architecture. By maintaining the affinity relation of a core and an NIC, software and hardware interrupts of the NIC are handled by the specified core. As a result, it incurs less cache miss and improves the execution efficiency of packet processing. However, as for packet forwarding, the NIC-based core affinity is not efficient. Since the receiving and sending of packet are handled in different cores usually, which causes the problems of mutex exclusion and cache coherence, Han et al. propose queue-based core affinity [22]. Each receiving and sending queue in an NIC maps to a core, and the corresponding CPU core accesses the queue exclusively, eliminating cache bouncing and lock contention caused by shared data structures. However, the NICs must support multiple queues and Receive Side Scaling (RSS), which can hardly be scalable.

OpenFlow Classification. As described in Figure 2(a), the OpenFlow classification in the kernel consists of the operations of packet parsing, Flow Cache lookup, counter updating, and action execution. Packet classification is time-consuming on general-purpose processors, and packet classification becomes even worse. According to the specification of OpenFlow 1.5, 41 fields of packet should be parsed, extracted, and looked up during the packet classification. That makes OpenFlow classification extremely complex and be of

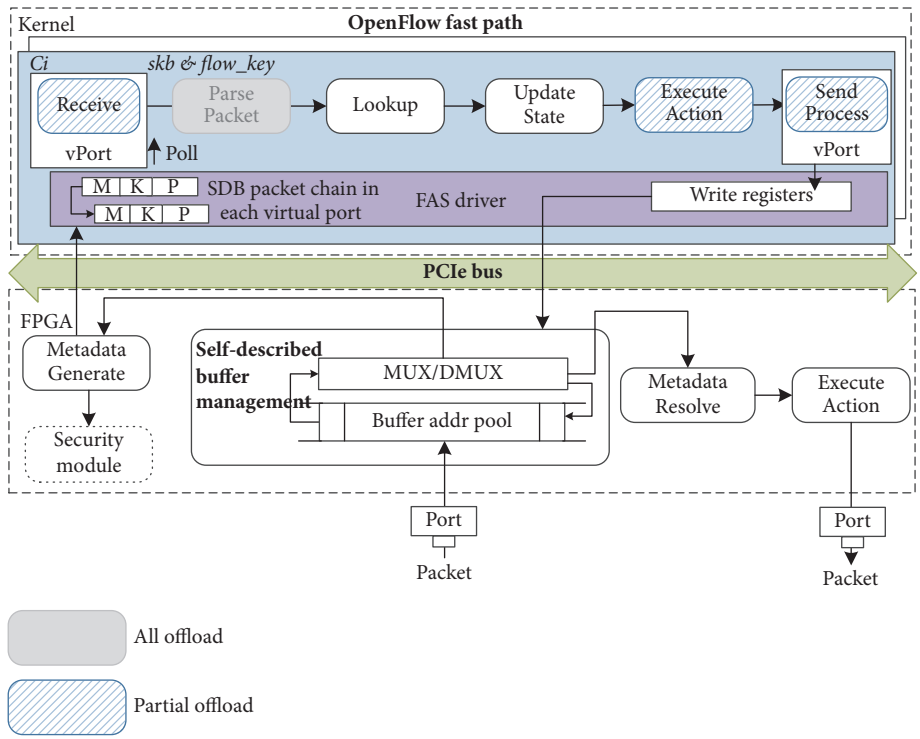


Figure 4: The framework of FAS mechanism.

low performance. Thus Putnam et al. designed three-layer cache architecture (microflow cache, megaflow cache, and OpenFlow pipeline) for accelerating OpenFlow classification [11]. Moreover, some actions of packet processing are costly in software, such as fields rewriting and packet encapsulation/decapsulation.

In addition, the performance of the SDN software switches will be largely decreased if complex security functions are required to be integrated. Most processing functions involved in security functions are stateful and CPU-consuming. The method to improve the security capacity without hurting performance should be investigated for SDN software switch.

4. FAS Mechanism

4.1. Architecture of FAS. Aiming at the above-mentioned bottlenecks of existing SDN software switches, we exploit programmable hardware, FPGA, to accelerate the rate of computation. FAS mechanism is designed to offload time-consuming functional modules in the OpenFlow software fast path to FPGA.

As depicted in Figure 4, the FAS mechanism consists of three components in FPGA. Self-Described Buffer Management module is used to offload some time-consuming parts in packet receiving and sending of the Linux kernel. Metadata Generate module is to offload the procedure of parsing packet. Execute Action is used to perform some actions after acquiring packet and its actions from Metadata Resolve module. Metadata Resolve module receives packet processing metadata information from software, resolves it, and notifies Execute Action module.

Packet Buffer Management Offloading. The management of Skb for each packet is a critical operation during packet I/O. It contains the operations of conversion from the raw packet to Skb, initialization of Skb, and allocation and deallocation of Skb. It consumes majority of CPU cycles in the procedure of packet I/O. Previously, our research group has proposed Self-Described Buffer (SDB) management in hardware to eliminate the packet buffer management expenditure in software [23]. In SDB, the original separated data structure Skb—packet and its metadata—is merged into a successive stored packet buffer (we call it the SDB packet buffer). The software preallocates fixed size space for SDB packet buffers in main memory at initial phase. This enables the SDB hardware to be able to dynamically allocate and recycle circular addresses of the SDB packet buffers during packet I/O. The packet buffer management overhead of software is thus eliminated. FAS makes use of SDB to alleviate the bottleneck of packet I/O.

Packet Parsing Offload. The OpenFlow fast path extracts matching fields to generate *flow_key* by parsing *Skb*. And the parsing procedure is the second most costly part as illustrated in Figure 3. We thus offload the operation of packet parsing to hardware and put the parsing result in the metadata of the packet. The parsing process is relatively straightforward in OpenFlow forwarding, as depicted in Figure 5. It can be easily implemented in FPGA hardware.

Action Execution Offload. The Execute Action module in FPGA executes partial actions for packet according to hardware capability. The fundamental operation is the forwarding action to corresponding port. In the sending process, the

Table 2: Comparisons of acceleration mechanisms for SDN software switches.

Names	Method	Acceleration object	Complexity	Network virtualization	HW state manage. cost
DPDK-based OF switching	Intel CPU/NIC	Packet I/O	Low	Support	No
Netmap-based OF switching	Netmap lib	Packet I/O	Low	Support	No
Flow Director	Driver modification	OF flow cache	Medium	Not support	High
SSDP	Commodity switching chip with TCAM	OF forwarding path	High	Not support	High
FAS	FPGA	Time-consuming functional modules in OF fast path	Medium	Support	Low

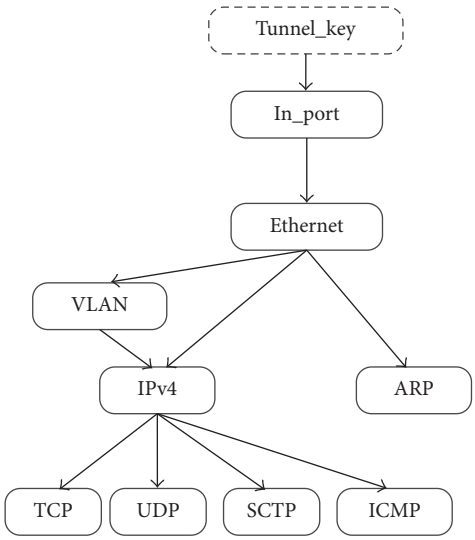


Figure 5: A parse graph example for OpenFlow.

hardware will resolve the packet address written in the FPGA registers and read the packet with metadata by DMA. The parameters of actions are carried in metadata, which are specified by the software. For example, if the queue action in metadata is resolved by the module of Metadata Resolve, the packet will be enqueued to the specified queue of corresponding port in FPGA. The offload of Execution Action can eliminate complex and time-consuming software packet operations, such as packet field rewriting and encapsulation/decapsulation.

Security Function Extension Interface. To support hardware security functions extension, we have proposed a well-defined module interface. The downstream and upstream interfaces of security modules are FIFO-liked interfaces and the control command is encoded into the metadata in front of the packet data. The security module can be easily embedded into the FPGA processing pipeline if the module interface definition is confirmed.

4.2. Comparisons between FAS and Existing Mechanisms. Many mechanisms have been proposed to improve the performance of OpenFlow software switching. We list state-of-the-art mechanisms and show their differences with FAS in Table 2. DPDK and netmap provide high-performance I/O framework for OpenFlow software switching. Note that DPDK relies on specific CPUs and NICs (e.g., Intel 82599), while netmap can be used for general NICs [24]. Flow Director proposes using the packet classification hardware on the NIC as OpenFlow fast forwarding path [25]. But the cost of maintaining the Flow Cache entries in NIC driver is relatively high. SSDP proposes enhancing slow software forwarding path with commodity switching chip including TCAM [26]. The common OpenFlow forwarding path is divided into two data planes in SSDP: macroflows in switch chip and microflows in CPU. Maintaining state consistency between the two data planes is intractable.

5. Preliminary Implementation

This section describes a design and preliminary implementation of FAS on NetMagic-Pro (NMP), which is an FPGA-based network processing platform with multicore CPU.

5.1. Brief Introduction of NetMagic-Pro. The hardware and software in NMP (as shown in Figure 6) are both programmable for packet processing. The total power consumption of NMP is 85 W. NMP consists of four parts: CPU board, FPGA board, line-card board, and power supply. The CPU board integrates an Intel i7-4700EQ CPU with 4 GB memory. The FPGA board is equipped with Altera EP4SGX180 and a piece of Flash storing the configuration file of the FPGA. The software running on multicore CPU communicates with FPGA through the PCIe bus. The PCIe v2.0 with 8 lanes offers a high link bandwidth of 40 Gbps in each direction. The line-card board provides eight 1-GigE Ethernet ports.

Similar to NetFPGA, NMP enables researchers and students to experiment with Gigabit rate networking hardware. The difference is that NMP provides better programmability in both hardware and software and higher software-hardware communication performance than NetFPGA. We implemented FAS mechanism on the platform of NMP.

5.2. FAS Driver. A kernel driver in the software is designed for FAS. It is responsible for NMP packets (including metadata, flow key, and packet data) communicating between the

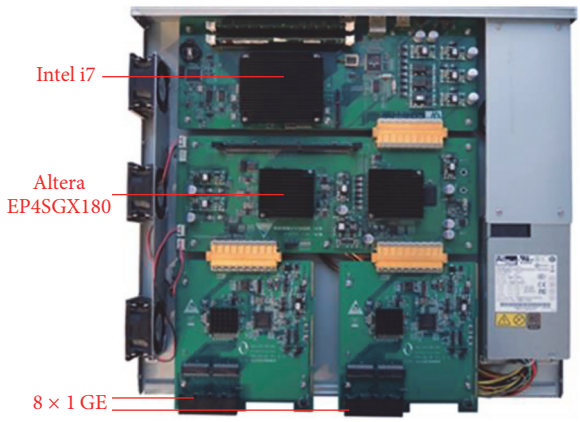


Figure 6: NetMagic-Pro network platform.

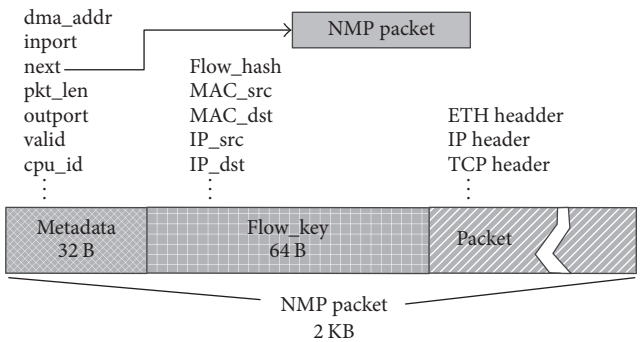


Figure 7: The format of NMP packet.

software and hardware. The data structure of NMP packet is illustrated in Figure 7. Instead of packet descriptors, all control messages for the packet are stored in metadata, including DMA address, ingress port, next NMP packet address, and packet valid indicator. *Flow_key* of 64 bytes consists of all matching fields extracted by hardware, whose size is 64 B. *Packet* accommodates the packet head and body, which is usually less than 1518 bytes. Therefore, 2 KB memory is preallocated for each NMP packet by software. FAS driver utilizes two techniques to optimize the performance of packet I/O and reducing cache miss rate in packet forwarding.

Polling Instead of Interrupt. Different from the hybrid of interrupt and polling in Linux NAPI (New API), we adopt polling approach to fetch the incoming NMP packets in FAS driver. It is reasonable since NMP is a packet forwarding platform. The FAS driver polls NMP packet from hardware DMA. The packet valid flag in the metadata is used to indicate whether the packet is ready to be processed. The NMP packets are organized into chain to simplify the polling. The next NMP packet address points to the next packet to be processed.

Core Affinity in Packet Dispatch. Each polling thread only runs on one CPU core. NMP packets are organized in chains; each chain is only assigned to one thread, namely, one core, for processing. With Run-to-Completion (RTC) mode of thread, each packet is received and transmitted by one core. It thus reduces the context switching overhead among different cores during processing. And it also reduces TLB update in accessing packet buffer.

Table 3: Basic functions for the virtual Ethernet port.

Interface of function	Description
static int nmp_open(struct net_device *netdev)	Open a port
static int nmp_close(struct net_device *netdev)	Close a port
static int nmp_xmit_frame(struct sk_buff *skb,struct net_device *netdev)	Send packet to a port
static struct net_device_stats *nmp_get_stats(struct net_device *netdev)	Acquire port statistics
static int nmp_set_mac(struct net_device *netdev,void *p)	Set MAC address of a port
static int nmp_change_mtu(struct net_device *netdev,int new_mtu)	Set MTU of a port

5.3. Virtual Ethernet Port. Virtual Ethernet ports are required for OpenFlow software switches to fully utilize the features of FAS. In NMP, the Ethernet ports are not standard commodity NICs. Therefore, we implement a customized network device for virtual Ethernet ports in NMP, which is applied to exchange packets and states for OpenFlow software switches.

The virtual Ethernet port provides basic functions for packet forwarding applications, including network device application, packets sending/receiving, port counting, MAC address configuration, and Maximum Transmission Unit (MTU) configuration. Thus, the virtual Ethernet ports of NMP transparently support all features of OpenFlow software switches running on NMP. The core operation of customized network device is the conversion between *Skb* data structure and NMP packet. As shown in Section 6.2, the cost of conversion is quite low.

There are three steps to implement the virtual Ethernet port for NMP. In the first step, we allocate a new network device by calling *alloc_etherdev* in the kernel. In the second step, we implement basic functions for the standard Ethernet port. The functions are listed in Table 3. All these functions are defined in the data structure of *net_device_ops*.

Among these functions, the most important one is packet sending, as shown in Pseudocode 1. There are two types of packets to be handled. For the packet received by FAS driver, if it is forwarded to some port; it will be converted from *Skb* to NMP packet. For the packet generated by the software, for example, OpenFlow messages sent to the controller, it is not provided with preallocated hardware-maintained addresses and other related information. It requires a new generated NMP packet with "soft" tag marked. Then the two types of the packets can be sent by writing hardware registers with corresponding metadata. The FPGA hardware executes operations on the packets according to their metadata.

In the third step, we register the new network device to the kernel. After configuring related parameters for the virtual

```
Pseudocode: Sending procedure in NetMagic-Pro with FAST
(1) nmp_xmit_Frame (Skb, nmp_netdev){
(2)    if (Skb->flag == NMP_PACKET){
(3)        nmp = transfer (Skb, nmp_netdev);
(4)        send_nmp_pkt (nmp){
(5)            //hardware send function by writing registers
(6)            NMP_SEND_PKT_REG(REG_addr,nmp.metedata);
(7)        }
(8)    }
(9)    else{
(10)       nmp = get_soft_nmp(Skb, nmp_netdev);
(11)       nmp.metedata.soft = 1;
(12)       send_nmp_pkt (nmp){
(13)           NMP_SEND_PKT_REG(REG_addr,nmp.metedata);
(14)       }
(15)   }
```

Pseudocode 1: Sending pseudocode for virtual Ethernet port.

Table 4: Components of platforms in experiment.

	NetMagic-Pro	Commodity PC
CPU	Intel i7-4700EQ 2.4 G	Intel i7-3770 3.4 G
The number of cores	4	4
Cache	6 MB L3-cache	6 MB L3-cache
DRAM	4 GB DDR3 SDRAM	4 GB DDR3 SDRAM
NIC	1 Gbps x8 NMP port	1 Gbps x2 Intel 82579
OS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS
FPGA	Altera EP4SGX180	—

Ethernet port, the *register_netdev* function is called in the kernel to complete the registration of the network device. Then the virtual Ethernet port can be accessed in the user space via the system command "ifconfig." At this time, we can create OpenFlow software switches on NMP. For example, the command "ovs-vsctl add-port br0 nmp1" is used to add port 1 in NMP to Open vSwitch's bridge.

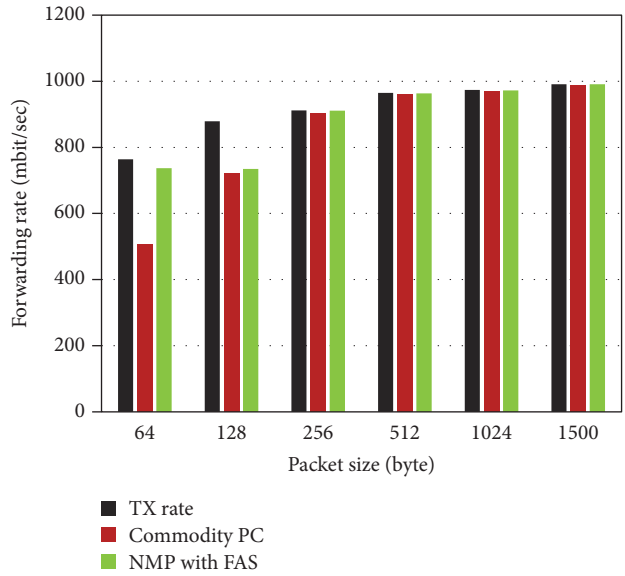


Figure 8: The comparison of forwarding rate with one rule.

6. Experiment Evaluation

6.1. Experiment Setup. We use Open vSwitch (version 2.3.1) [9] as the typical SDN software switch in experiments. Open vSwitch runs on NMP with FAS enabled and a commodity PC. Both platforms are directly connected to IXIA XM2 Tester with two ports/NICs. The two ports in IXIA XM2 Tester are used for traffic source and sink, respectively. The packet size of testing flows can be configured in IXIA XM2 Tester.

Table 4 lists the specification of the experimental components. Both devices under test have almost the same hardware configuration except for the CPU. Note that the performance of the PC's CPU (Intel i7-3770) is a little better than that of NMP (Intel i7-4700EQ). In the experiments, we mainly evaluate the efficiency of FAS mechanism from two aspects: forwarding rate and forwarding latency. We believe that PCIe bandwidth for transmitting packets between the software and FPGA is not a bottleneck, because PCIe 2.0 x8 links provide 40 Gbps, which is far more enough than required for unidirectional traffic generated in the experiment.

6.2. Forwarding Performance Evaluation. Forwarding performance of OpenFlow software switching is concerned with packet size and flow rules. Firstly, we test the forwarding rate of Open vSwitch under 1 flow rule. All tests last 60 seconds and forwarding rates are sampled by 1 second. The average forwarding rates are shown in Figure 8. When forwarding traffic is with the size of 64 B Ethernet packets, NMP with FAS achieves throughput of 740 Mbps, about 97% of the theoretical wire-speed (762 Mbps). NMP with FAS gets 44% higher performance than the commodity PC. When the packet size is 128 B, the forwarding rate of NMP with FAS achieves wire-speed forwarding, which is 18.5% higher than the commodity PC. The gap of performance between FAS and

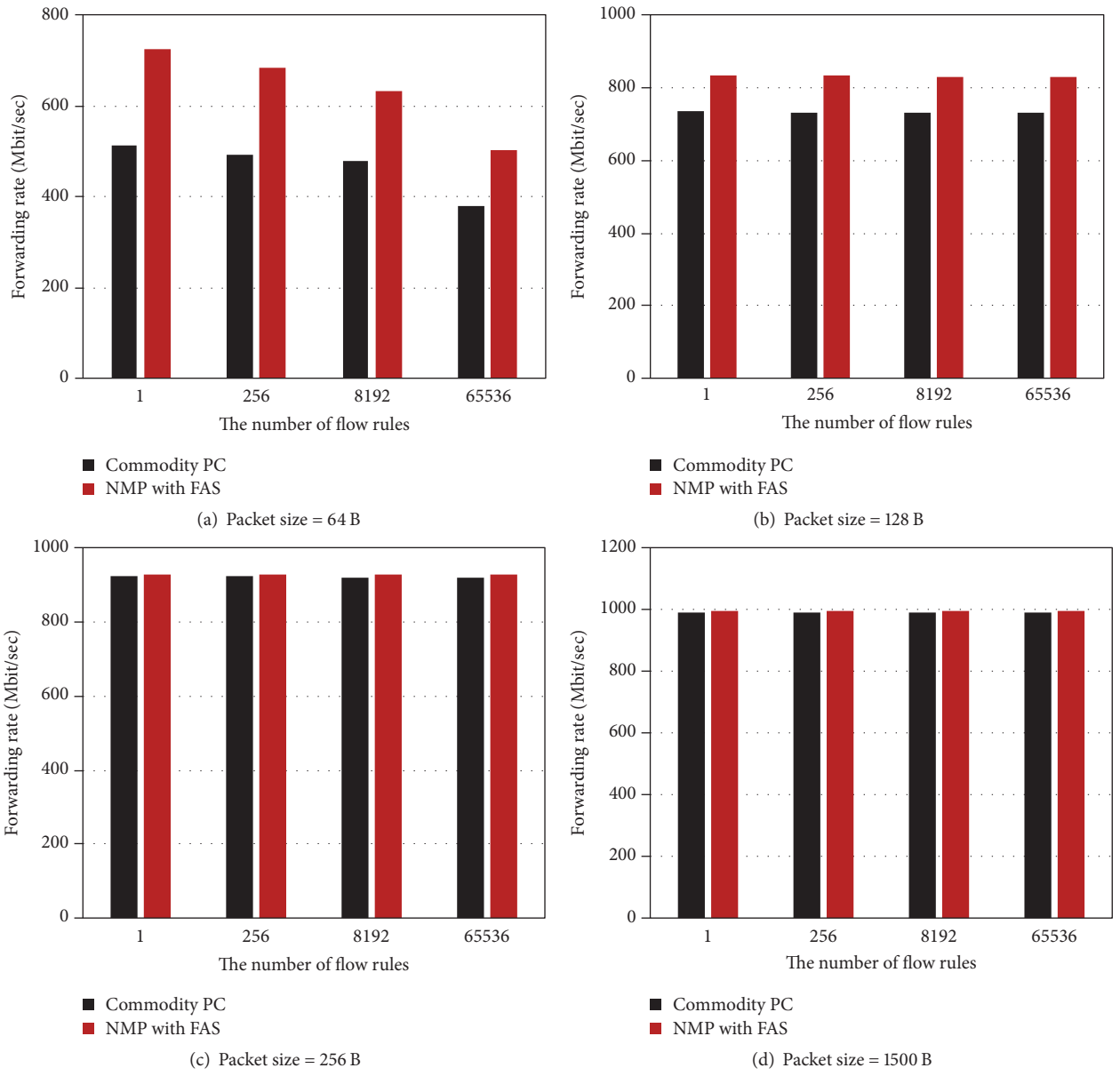
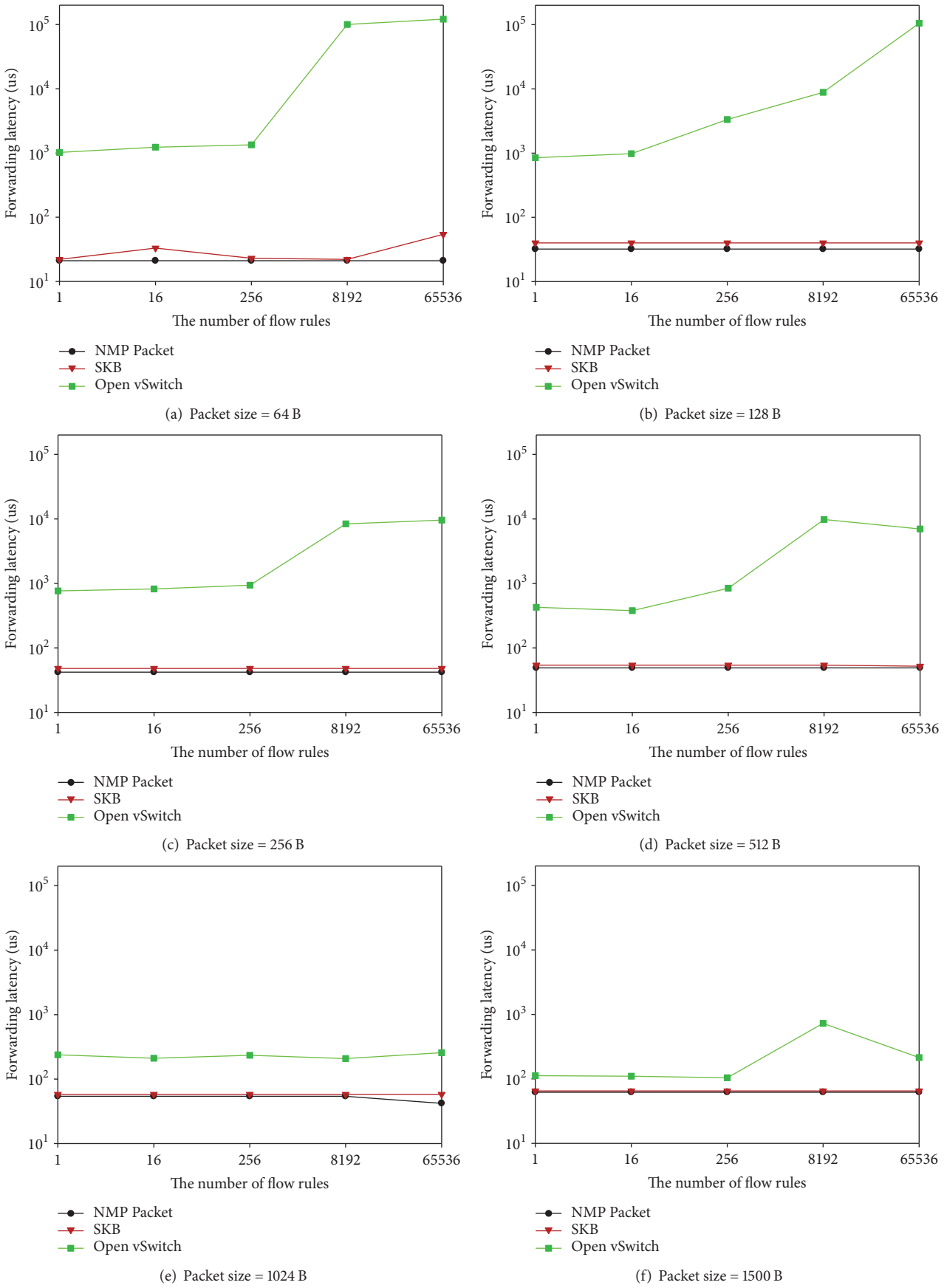


Figure 9: The forwarding rate with various number.

the original OVS becomes smaller when the size of packets increases. That is because both can achieve wire-speed for large packets. We can conclude that, in the case of 1 flow rule, NMP with FAS can get nearly wire-speed forwarding rate even for minimum size packets, which significantly outperforms the commodity PC.

Secondly, we make comparisons of the forwarding rate under different number of flow rules. The preinstalled flow rules are all mutually exclusive with others. The numbers of rules are 256, 8192, and 65536 in different experiments. The traffic is generated according to the rules to guarantee that every packet matches the corresponding flow rule. We also vary the size of packets. The experimental results are shown in Figure 9. For the cases of 512 B, 1024 B, and 1500 B Ethernet packet, the forwarding rates of NMP with FAS and the commodity PC have little difference and both approach the wire-speed under all different numbers of flow rules. The data of 512 B and 1024 B are omitted for simplicity. For the packet size of 64 B (Figure 9(a)), the forwarding rate goes down with increasing flow rules. When the number of flow rules reaches 65536, the forwarding rates of NMP with FAS and the commodity PC fall by 45% and 47%, respectively, when compared to one-rule case, but the forwarding rate of the NMP with FAS is still 44% higher than that of the commodity PC. When the packet size is 128 B (Figure 9(b)), with the number of flow rules growing, the forwarding rates of NMP with FAS and the commodity PC both vary little. And NMP with FAS is 18% higher than the commodity PC. In summary, the FAS can provide higher forwarding rate, especially for small-size packets.

We evaluate the forwarding latency with different packet sizes and different number of flow rules. As Figure 10 shows,



(a) Packet size = 64 B

(b) Packet size = 128 B

(c) Packet size = 256 B

(d) Packet size = 512 B

(e) Packet size = 1024 B

(f) Packet size = 1500 B

Figure 10: Comparison of forwarding latency with various packet sizes.

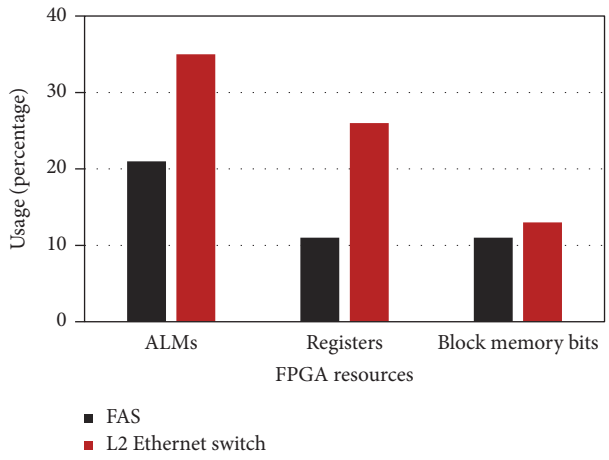


Figure 11: The FPGA resource usages of FAS and L2 Ethernet switch.

in the experiment, Skb forwarding indicates packet forwarding with the conversion operation between NMP packet and Skb data structure. It can be observed that the latency of Skb forwarding on NMP is close to the latency of the NMP packets. That means the cost of conversion operation between NMP packet and Skb is quite small. With the growing of packet size, the latency gap between Skb forwarding and Open vSwitch is decreasing. That is because if the packet size is small, there will be more packets in the software processing queue. The packets will experience longer delay when queuing. That incurs large forwarding latency.

At last, we compare the implementation complexity of FAS to a standard L2 Ethernet switch. The consumption of FPGA resource is depicted in Figure 11. As we can observe, the logic utilization of FAS in ALMs is about 40% less than the L2 Ethernet switch. The resource usage results suggest that the implementation of FAS is simple and feasible in FPGA.

We can conclude from the experiment that FAS provides considerable acceleration for OpenFlow software forwarding, especially for small packets. And the implementation complexity in FPGA is acceptable.

7. Conclusions

The SDN switches are the fundamental infrastructure to supply flexible control of flows. SDN software switches running on commodity multicore platforms are widely deployed due to their upgradability, programmability, and low cost. However, the forwarding performance as well as security capacity provided by general-purpose SDN software switch platform is usually not satisfied. The case becomes even worse for OpenFlow forwarding.

In this paper, we design and implement an FPGA-based mechanism accelerating and securing SDN software switches, namely, FAS. FAS provides a framework to offload the time-consuming modules and real-time security modules of SDN software switch and it employs some optimization techniques for solving the performance bottleneck between software and FPGA hardware. Our experimental results show that FAS utilizes reasonable FPGA resources and outperforms commodity platforms with nearly 44% higher forwarding rate for small packets. FAS can also be used to enhance the security of SDN software switches by allowing the bump-in-the-wire security modules to be integrated in FPGA.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by a research grant from Chinese National Programs for Key Technology of Software Definition Network (SDN) Supporting Resource Elasticity Scheduling and Equipment Development (863 Programs) (no. 2015AA016103) and National Natural Science Foundation: Synergy Research on CPU/FPGA Heterogeneous Network Processing System for Complex Network Applications (no. 61702538). The authors thank Dr. Jianbiao Mao and other helpful friends for great help.

References

[1] *Software-Defined Networking: The New Norm for Networks, ONF White Paper*, http://www.opennetworking.org.

[2] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[3] *OpenFlow Specification 1.5.1, Open Networking Foundation*, 2015, http://www.opennetworking.org.

[4] P. Bosshart, G. Gibb, H.-S. Kim et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '13)*, pp. 99–110, August 2013.

[5] R. Ozdag, *Intel Ethernet Switch FM6000 Series Software Defined Networking*, August, Intel Corporation, 2012.

[6] The OpenFlow Consortium, *Openflow Switching Reference System*, January 2011, http://www.openflowswitch.org.

[7] "OFSoftSwitch13," http://cpqd.github.com/ofsoftswitch13.

[8] Y. Mundada, R. Sherwood, and N. Feamster, "An OpenFlow switch element for Click , in Symposium on Click Modular Router," 2009.

[9] *Open vSwitch – An Open Virtual Switch*, September 2014, http://www.openvswitch.org.

[10] Z. Cai, Z. Wang, K. Zheng, and J. Cao, "A Distributed TCAM coprocessor architecture for integrated longest prefix matching, policy filtering, and content filtering," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 417–427, 2013.

[11] A. Putnam, A. M. Caulfield, E. S. Chung et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, pp. 13–24, IEEE, Minneapolis, Minn, USA, June 2014.

[12] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*, pp. 151–156, USA, August 2014.

[13] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08)*, pp. 1–9, USA, November 2008.

[14] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi, "Cheap silicon: A myth or reality? Picking the right data plane hardware for software defined networking," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*, pp. 103–108, China, August 2013.

[15] B. Pfaff, J. Pettit, T. Koponen et al., "The design and implementation of open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pp. 117–130, USA, May 2015.

[16] A. Tirumala, F. Qin, J. Dugan et al., *iPerf: TCP/UDP Bandwidth Measurement Tool*, 2008.

[17] OProfile, http://oprofile.sourceforge.net.

[18] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA '11)*, pp. 255–265, USA, February 2011.

[19] Intel, "Intel data plane development kit (intel DPDK)," in *Programmer's Guide*, October 2013.

[20] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proceedings of the USENIX Annual Technical Conference*, pp. 101–112, 2012.

[21] K. Subramanian, L. D'Antoni, and A. Akella, "Genesis: Synthesizing forwarding tables in multi-tenant networks," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, pp. 572–585, France, January 2017.

[22] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proceedings of the 7th International Conference on Autonomic Computing (SIGCOMM '10)*, pp. 195–206, India, September 2010.

[23] L. Tang, J. Yan, Z. Sun, T. Li, and M. Zhang, "Towards high-performance packet processing on commodity multi-cores: current issues and future directions," *Science China Information Sciences*, pp. 1–16, 2015.

[24] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '12)*, pp. 2471–2479, USA, March 2012.

[25] V. Tanyingyong, M. Hidell, and P. Sjodin, "Using hardware classification to improve PC-based OpenFlow switching," in *Proceedings of the 2011 IEEE 12th International Conference on High Performance Switching and Routing (HPSR '11)*, pp. 215–221, Spain, July 2011.

[26] R. Narayanan, S. Kotha, G. Lin et al., "Macroflows and microflows: Enabling rapid network innovation through a split SDN data plane," in *Proceedings of the 1st European Workshop on Software Defined Networks (EWSDN '12)*, pp. 79–84, Germany, October 2012.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 6361901, 12 pages
https://doi.org/10.1155/2018/6361901

Research Article

Kuijia: Traffic Rescaling in Software-Defined Data Center WANs

Che Zhang, Hong Xu, Libin Liu, Zhixiong Niu, and Peng Wang

NetX Lab, City University of Hong Kong, Kowloon Tong, Hong Kong

Correspondence should be addressed to Che Zhang; czhang226-c@my.cityu.edu.hk

Received 27 September 2017; Accepted 6 November 2017; Published 15 January 2018

Academic Editor: Yang Xu

Network faults like link or switch failures can cause heavy congestion and packet loss. Traffic engineering systems need a lot of time to detect and react to such faults, which results in significant recovery times. Recent work either preinstalls a lot of backup paths in the switches to ensure fast rerouting or proactively prereserves bandwidth to achieve fault resiliency. Our idea agilely reacts to failures in the data plane while eliminating the preinstallation of backup paths. We propose Kuijia, a robust traffic engineering system for data center WANs, which relies on a novel failover mechanism in the data plane called rate rescaling. The victim flows on failed tunnels are rescaled to the remaining tunnels and enter lower priority queues to avoid performance impairment of aboriginal flows. Real system experiments show that Kuijia is effective in handling network faults and significantly outperforms the conventional rescaling method.

1. Introduction

Traffic engineering (TE) is increasingly implemented using software-defined networking (SDN), especially in inter-data center WANs. Examples include Google's B4 and Microsoft's SWAN [1, 2]. Usually, some tunnel protocol is used: the controller establishes multiple tunnels (i.e., network paths) between an ingress-egress switch pair and configures splitting weights at the ingress switch. The ingress switch then uses hashing based multipath forwarding such as ECMP to send flows.

An important issue about TE that is commonly overlooked in the literature is robustness against failures. In reality, failures are the norm rather than exception, especially for large networks. Table 1 shows failure statistics data from Microsoft's data center WAN [3]. The probability of having at least one link failure within five minutes, which corresponds to the TE frequency [1, 2], is more than 20%. Even with a single link failure, the impact can be severe as a data center WAN operates near capacity for maximum efficiency [1, 2].

Controller intervention offers the best failure recovery performance given its global network view. However, recomputing a new TE plan and updating the forwarding rules across the entire network take at least minutes and are error-prone [4–6]. When the controller is being attacked by Distributed Denial of Service, or others, the reaction time of the controller can be even longer [7–9]. Therefore, we need to have a mechanism to protect the data plane from congestion after failures without the intervention of a controller. For responsiveness, a simple data plane reactive method called *rescaling* is deployed in practice. Upon detecting the failure, the ingress switch normalizes splitting weights to redirect traffic among the remaining tunnels [6]. Rescaling quickly restores connectivity without involving the controller at all. However, since traffic is still sending at the original rates, local rescaling more than often leaves the network in a congested state [6].

Some solutions have emerged to solve this practically important issue. Suchara et al. [10] propose to precompute the splitting weights for arbitrary faults to reduce transient congestion. This approach may not work well for large production networks due to the exponentially many failure cases. Liu et al. [6] propose forward fault correction (FFC). FFC proactively considers failures when formulating the TE problem. As a result, the TE solution can guarantee no congestion happens for arbitrary k faults with rescaling. Intuitively, such strong guarantees come with a price: in FFC, a portion (about 5%–10% depending on k) of the network capacity has to be always left vacant in order to handle traffic from rescaling. This means hundreds of Gbps bandwidth is

Table 1: Link failure frequencies in Microsoft data center WAN [3].

Number of link failures	Time intervals		
	2 min	5 min	10 min
1	10.6%	21.5%	31.2%
2	0.14%	1.1%	4.2%
3	0.14%	0.7%	1.4%

wasted most of the time. Arguably, the cost outweighs the benefits of eliminating transient congestion.

Thus, the following question remains largely open: can we design a robust TE system that is (1) responsive in quickly restoring connectivity, (2) effective in reducing congestion without excessive bandwidth overhead, and (3) practical and simple enough to be deployed in existing switches?

Our main contribution is the design and evaluation of Kuijia (the word "Kuijia" means armor in Chinese; Kui is for protecting the head and neck, and Jia is for protecting the torso), a robust TE system for data center WANs that answers the above question affirmatively. We argue to isolate the affected flows from the aboriginal ones to avoid the propagation of failure impact. This is particularly useful when there are many latency-sensitive flows like video, online shopping, and search whose traffic is rapidly growing due to the development of mobile devices and high-speed cellular networking.

Kuijia relies on a novel failover mechanism in the data plane called *rate rescaling* that rescales the traffic sending rates in addition to splitting weights, by using priority queueing at switches. The victim flows are still rescaled to the remaining tunnels, but they now enter a lower priority queue at the switches and do not compete with aboriginal flows on the remaining tunnels. Effectively, their sending rates are automatically throttled to only using the available bandwidth of the remaining tunnels without the need for controller intervention.

Kuijia with rate rescaling offers an advantage over simple rescaling. Rescaling only ensures the failed link is avoided. Yet, flows are still sending at their original rates to the remaining tunnels. Clearly, with the loss of capacity, many packets will be dropped after rescaling, and every TCP flow on the remaining tunnels will back off and suffer from throughput loss. Rate rescaling ensures there is no congestion even with the victim flows, and the aboriginal flows are not affected. It maintains the responsiveness of rescaling, is simple to implement as priority queueing, is widely supported by commercial switches, and is effective in utilizing the available bandwidth due to the work-conserving nature.

This paper is an extended version of work published in [11]. We extend our previous work to handle traffic with multiple priorities. Specifically, we propose a new flow table decomposition method to produce multiple tables in order to reduce the number of flow entries. For experiment, we add a large-scale simulation to demonstrate our design of using original weights for rate rescaling is more simple and effective compared with storing and using the precomputed weights. In addition, we add testbed experiments to evaluate performance and flow entries' memory usage of Kuijia for multipriority traffic. We also add a new section to discuss several issues related to the use of Kuijia in a production data center WAN and we explain them in three aspects, which include traffic characteristics, traffic priorities, and impact of flow size to hashing.

2. Related Work

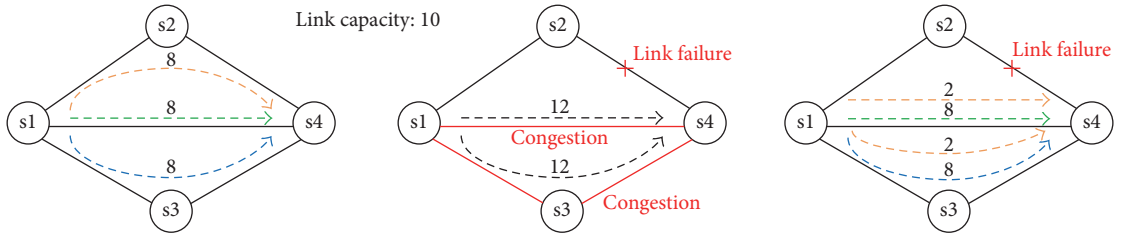
Failures in SDN. There is much work to deal with failures in SDN. New abstractions are proposed in [12, 13] to enable developers to write fault-tolerant SDN applications. Some other work relies on the local fast failover mechanism introduced in OpenFlow to design new functions. Schiff et al. [14] propose SmartSouth to provide a new data plane for OpenFlow switches that can implement fault-tolerant mechanisms. Borokhovich et al. [15] develop algorithms to compute failover tables. Chang et al. [16] develop an optimization-theoretic framework to validate network designs under uncertain demands and failures. Kuijia is different in that it focuses on remedying the congestion due to rescaling.

Failures in Data Center WANs. The most widely used approach to deal with network failures, including link or switch failures, is to recompute a new TE solution based on the changed topology and reprogram the switches [1, 2]. However, such a reactive approach is not fast and efficient enough as discussed in Section 1.

Several proactive approaches have been proposed to solve this important problem. Suchara et al. [10] modify the ingress switches' rescaling behavior. Rather than simple proportional rescaling, tunnel splitting weights are based on the set of residual tunnels after the failure. These weights are precomputed and preconfigured at switches. Although it achieves near-optimal load balancing, this approach can handle only a limited number of potential failure cases as there are exponentially many of them to consider, and switches have limited space for flow rules.

Liu et al. [6] propose forward fault correction (FFC) to handle failures proactively. FFC ensures that each time the operator computes a new TE, it is congestion-free not only without any failures, but also with any link failures that could happen in the following TE interval. This is in sharp contrast to recomputing TE after failures as it requires no update to TE in response to failures. Although FFC spreads network traffic such that congestion-free property is guaranteed under arbitrary combinations of up to k failures, the price is very high. About 5%–10% of the network capacity depending on k has to be always left vacant to handle traffic from rescaling. SWAN [1] develops a new technique that also leverages a small amount of scratch capacity on links to apply updates in a provably congestion-free manner.

Instead of waiting for rescaling in the ingress switches, Zheng et al. [17] use backup tunnels that start from the failing switch and end at the egress switches to redirect the affected traffic. It can be faster and more effective in reducing congestion but the cost is still high as there are exponentially many failure cases to consider. The large number of flow entries required for backup tunnels is too expensive for the limited hardware tables [18].



(a) Initial traffic distribution (b) Traffic distribution after rescaling (c) Traffic distribution after rate rescaling

FIGURE 1: Comparison of rescaling and rate rescaling in handling a single link failure.

Our method, Kuijia, is different from the existing work as we use priority queueing in switches, which is simple to implement in practice and has no overhead of excessive bandwidth or a large number of flow entries for backup tunnels.

3. Motivation

We motivate our idea using a simple example. Figure 1(a) shows a small network with traffic sent from s1 to s4. The traffic is routed over three tunnels: s1 → s2 → s4 (T1), s1 → s4 (T2), and s1 → s3 → s4 (T3). Each tunnel is configured with the same weight and carries 8 Gbps traffic. When link s2–s4 in T1 fails, s1 rescales the traffic to the remaining two tunnels, resulting in traffic distribution as shown in Figure 1(b). Since the traffic is still sent at 24 Gbps, the remaining tunnels T2 and T3 need to carry 12 Gbps each and are heavily congested.

The difference between Kuijia and conventional rescaling is that Kuijia differentiates aboriginal traffic on remaining tunnels from victim traffic rescaled to them. Kuijia places the victim traffic into a low priority queue of the remaining tunnels, while the aboriginal traffic enters a higher priority queue. With Kuijia, traffic is distributed as shown in Figure 1(c). The victim traffic (shown in yellow) uses the remaining capacity of T2 and T3 and sends at 2 Gbps in each tunnel. This does not cause any congestion or packet loss for the aboriginal flows and fully utilizes the link capacity.

We experimentally verify the effectiveness of Kuijia using a testbed on Emulab [19]. We connect 4 Emulab servers running OpenvSwitch (OVS) [20] to form the same topology as in Figure 1. A dedicated server runs the controller to manage the network. In Kuijia, 3 strict priority queues are configured on the egress ports of each switch. Control messages enter the highest priority queue with priority 0. Normal application traffic has priority 1 but is demoted to priority 2 once it is rescaled to other tunnels after failures. For simplicity, both rescaling and Kuijia are implemented in the control plane: a switch informs the controller of a link failure. The controller then adjusts the flow splitting weights and priority numbers at the corresponding ingress switches of the victim flows.

Switch s1 starts `iperf` TCP connections to s4 over three tunnels. Since in our example rescaling splits the victim flow on T1 to T2 and T3, we configure s1 to send two `iperf` TCP flows f1 and f2 over T1. Flow f1 is rescaled to T2 and f2 to T3. s1 sends another two flows f3 and f4 over T2 and T3, respectively.

TABLE 2: Testbed experiment for the motivation example, where the remaining tunnels have a vacant capacity for victim traffic.

Flows	f1	f2	f3	f4
Rescaling				
Before failure	380 Mbps	381 Mbps	762 Mbps	762 Mbps
After failure	379 Mbps	378 Mbps	584 Mbps	586 Mbps
Kuijia				
Before failure	380 Mbps	381 Mbps	762 Mbps	762 Mbps
After failure	177 Mbps	177 Mbps	762 Mbps	762 Mbps

TABLE 3: Testbed experiment for the motivation example, where the remaining tunnels do not have a vacant capacity.

Flows	f1	f2	f3	f4
Rescaling				
Before failure	475 Mbps	468 Mbps	943 Mbps	941 Mbps
After failure	472 Mbps	474 Mbps	465 Mbps	470 Mbps
Kuijia				
Before failure	475 Mbps	468 Mbps	943 Mbps	941 Mbps
After failure	0.074 Mbps	0.011 Mbps	943 Mbps	941 Mbps

We run two experiments with different extents of congestion to demonstrate the effectiveness of Kuijia. Table 2 shows the result when flows f3 and f4 send at 800 Mbps and f1 and f2 send at 400 Mbps each before failure. This represents the case when the remaining tunnels (T2 and T3) have vacant capacity. We observe that, with simple rescaling, the throughput of all flows degrades after failures, since the aggregate demand of victim and aboriginal flows (1.2 Gbps) exceeds 1 Gbps. Now with Kuijia, aboriginal flows f3 and f4 are not affected at all as shown in Table 2, and the victim flows use the vacant capacity of 200 Mbps without causing any congestion or packet loss.

Table 3 shows the result when f3 and f4 send at 1 Gbps and f1 and f2 send at 500 Mbps each before failure. This represents the case when the remaining tunnels do not have any capacity for the victim traffic. Rescaling again causes severe congestion to aboriginal traffic on the remaining tunnels, and after TCP convergence f3 and f4 achieve throughput of ~470 Mbps. With Kuijia, the victim traffic (f1 and f2) does not obtain any throughput and the aboriginal flows are not impacted at all.

4. Design

In this section, we first introduce the background of TE and rescaling implementation in production data center WANs,

and then we explain the design of Kuijia and its difference from rescaling.

4.1. Background. In a data center WAN, after the controller computes the bandwidth allocation and weights for all the tunnels of each ingress-egress switch pair, it issues the group table entries and flow table entries in OpenFlow [1, 21]. Label-based forwarding is usually used to reduce forwarding complexity [2]. The ingress switch uses group entry in the group table to split traffic across multiple tunnels and assigns a label to traffic of a specific tunnel. The downstream switches simply read the label and forward packets based on the flow entries for that label from the flow table. As an example, Figure 2 shows the group tables and flow tables of four switches for the network used in Figure 1. The forwarding label can be MPLS, VLAN tags, and so forth.

Flows are hashed to different tunnels consistently (and different labels are applied) when they arrive at the ingress switch for simplicity. Thus, splitting weights are configured as ranges of the hashed values. For example, in Figure 3(a), the weights are 0.5, 0.3, and 0.2 for tunnels T1, T2, and T3, respectively. For simple rescaling, its implementation is as follows. Suppose tunnel T1 fails as in the motivation example. The ingress switch rescales the traffic to the remaining tunnels by removing the bucket in the group entry that corresponds to the failed tunnel as shown in Figure 2 (entries with $*$ only exist in Kuijia, not in rescaling). The entries in the blue table are issued after failures. In addition, since T1 fails, the hash value ranges for T2 and T3 also "rescale" accordingly, so that weights of T2 and T3 are now 0.6 and 0.4. As discussed already, this may cause congestion after rerouting the victim traffic [6].

4.2. Kuijia. Here, we explain the detailed design of Kuijia for SDN based data center WANs. We focus on dealing with single link failures, which are most common in production networks as shown in Table 1. Multiple link failures are rare and can be handled by controller intervention on a need basis.

We propose Kuijia with *rate rescaling* to reduce the impact of congestion after failures. Its design is simple and can be implemented in OpenFlow switches. Suppose there are k tunnels for traffic between a given ingress-egress switch pair, and one tunnel fails. Kuijia keeps the original hash range and separates the hash range of the failed tunnel into $k-1$ parts according to weights of the $k-1$ tunnels to form the new hash ranges. It also marks the hash range of the failed tunnel to low priority in order to enforce priority queueing. This way, Kuijia can differentiate the aboriginal traffic on the remaining $k-1$ tunnels from the victim traffic that is rescaled to them. For the same example in Figure 3(b), when T1 fails, its hash range is split into two parts for T2 and T3 with weights to 0.3 and 0.2, respectively. One can easily verify that the aboriginal flows on T2 and T3 are still hashed to the same ranges and routed normally. Victim traffic on T1 is now rescaled to T2 and T3 and tagged as low priority in order not to affect the aboriginal flows.

In order to verify that it is effective to use the original weights and just separate the hash range of the failed tunnels, we compare it to rerouting using precomputed weights. The precomputation works as follows. For a given link failure, we keep the weights of unaffected tunnels and sending rates of unaffected flows unchanged. We take the remaining bandwidth of each link and tunnel, as well as the victim flows that need to be rerouted, as the input of a new TE program and compute the best weights for these victim flows. Clearly, recomputing the weights yields the optimal performance to deal with the failure. We run experiments with 10 random graphs each having 100 nodes and 200 links. For each graph, we randomly select 40 switch pairs for 10 runs, and each pair has 3 tunnels. As [10] shows, even in a large ISP backbone, three or four tunnels per switch pair is sufficient. In each run, we vary the demand of each switch pair from 0.8 Gbps to 3.0 Gbps. We sequentially fail all the edges one by one and then compute the average throughput loss for each demand.

Figure 4 shows the comparison result. We observe that the performance of using original weights is highly comparable to that of precomputing new weights. As each switch pair has 3 tunnels and they may have some common links, with one link down, there is only one or two remaining tunnels for each affected switch pair. When the demand is small, which means the network is not that congested before failure, simply using original weights can meet the demand in most cases. When the demand becomes larger, which means the network is more congested, using TE to precompute weights cannot reduce throughput loss much further. Reference [10] also shows similar results. The results demonstrate that our design of using original weights and separating the hash range is simple and effective and also avoids the complexity of storing the precomputed weights.

Note that when one link fails, any intermediate switch may potentially become congested due to rescaling. Thus, it is necessary for *all* switches to perform priority queueing for the victim flows, not just the corresponding ingress switch. To do that, there are two ways. The first one is to compute which links will be congested after rescaling, and then we only need to configure the corresponding flow entries at those switches to realize priority queueing. Although this method uses fewer flow entries, it is hard to achieve in reality because the ingress switch has no information of all the traffic in the network, and the controller has to compute which links will be congested after failures actually happen, which defeats the purpose of having a data plane failover mechanism.

Thus, Kuijia uses a simple method that doubles the flow entries in all switches for each tunnel. We have a normal priority queue and a low priority queue at each port of each switch. Each queue has the same set of flow entries to route traffic. Traffic with low priority tags is sent to the low priority queue as shown in Figure 5. This is simple to implement in the data plane and can handle any link failures quickly.

For example, in Figure 2, for aboriginal flows sending to 10.0.2.0/24, they match `low` = 0, `inport` = 1, `pathid` = 3 in s3 and go to `queue`(1). The corresponding entry, matching `low` = 1, `inport` = 1, `pathid` = 3 will go to `queue`(2) which is the low priority queue and is used when there are victim flows due to link failure, for example, when the s2–s4 link is down. In the ingress switch s1, the group table applies low priority tags to the victim flows (entries with $*$) and directs the packets to the `outport` which is connected to the next-hop switch.

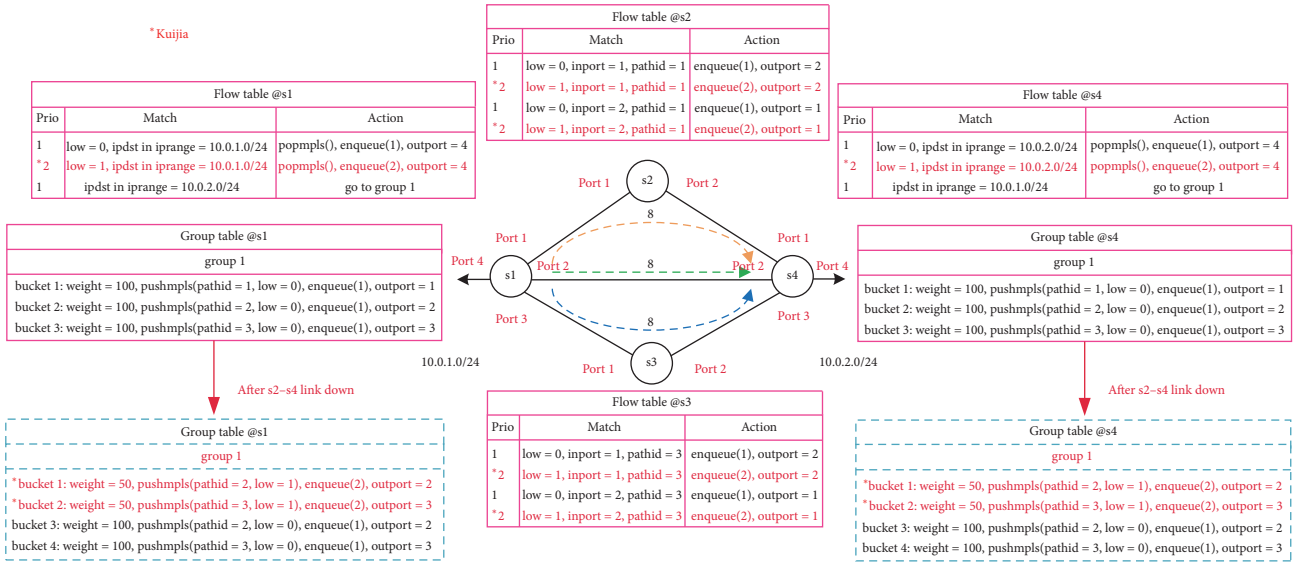


Figure 2: The design of flow table and group table of each switch in the simple topology.

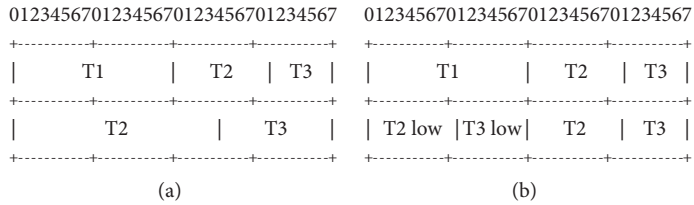


Figure 3: The change of hash range after failure.

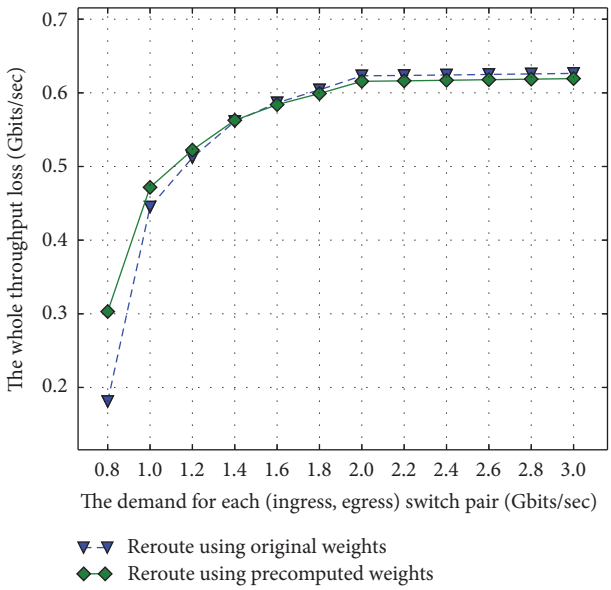


Figure 4: The whole throughput loss after single link failure as demand increases.

Each intermediate switch of the tunnel matches packets on priority, inport, and pathid. Victim flows are then routed to queue(2) (rerouting flow, low priority tag = 1) of outport while aboriginal flows are routed to queue(1).

Note that as TCP connection needs two-way communication, the flow entries and group entries are also issued for two-way communication. For example, if s1 sends TCP packets to s4 through s1-s3-s4, we need to issue the flow entries not only for the direction of s1 → s3 → s4, but also for the reverse direction of s4 → s3 → s1 (e.g., s3 has the flow entry: match{low = 0, inport = 2, pathid = 3}, actions{enqueue(1), outport = 1}). Hence, the TCP ACK packets could be returned to s1 by matching the flow entry of switches in the reverse direction. We use MPLS `label` field to store our path ID (each tunnel (path) has a unique path ID) and `tc` field to store our low priority tag (0 means aboriginal flow and 1 means victim flow).

4.3. Multiple Priorities. Kuijia can also be extended to handle traffic with multiple priorities. We have assumed that all traffic has the same priority before failures thus far. In practice, it is common for networks to use multiple priorities to differentiate applications with distinct performance requirements [22, 23]. Kuijia can be extended to this setting so that high priority victim flows could also obtain as much bandwidth as possible after rerouting.

To illustrate Kuijia's working for multipriority traffic, consider a simple case where switches in the network have

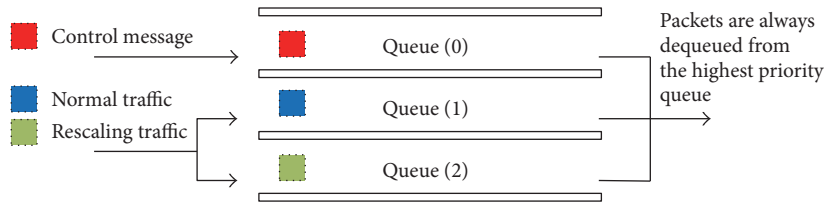


Figure 5: The switch queues in Kuijia.

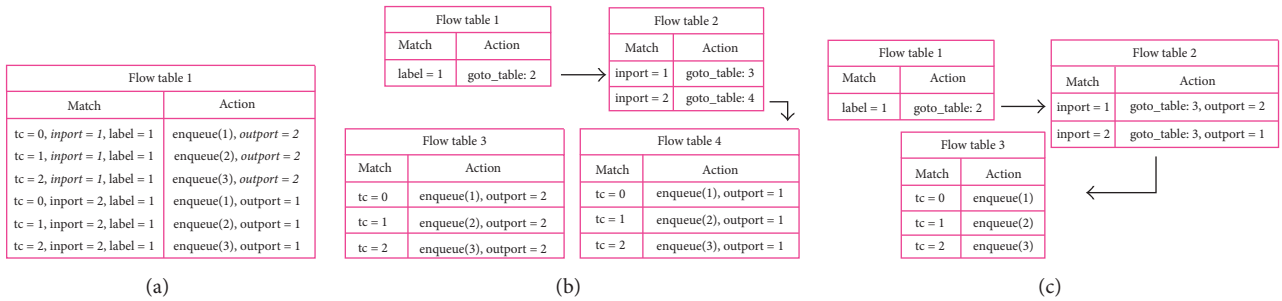


Figure 6: The example of three kinds of design for flow entries under multiple priorities.

8 priority queues each. The highest priority 0 is for control traffic. We set priorities 1 to 4 for high priority traffic, while we set priority 6 for background low priority traffic before failures. After a link failure, the high priority victim flows are rerouted by Kuijia to the remaining tunnels through priority 5 to guarantee they can get bandwidth by occupying the bandwidth of low priority traffic. Similarly, the low priority victim flows are rerouted to the remaining tunnels through priority 7 which is the lowest priority to guarantee that the aboriginal flows are not affected.

Note two design particulars here. First, Kuijia ensures that the lowest priority traffic is always dropped first when there is not enough bandwidth. Second, the bandwidth of each high priority queue can be changed by occupying the bandwidth of lower priority.

Therefore, we set the maximum bandwidth for the priority `queue`(0) to `queue`(6) to the capacity of each link and the minimum bandwidth for the lowest priority `queue`(7) to `queue`(0).

One challenge with multiple priorities is how to handle the increased number of flow entries required to implement Kuijia. Using a simple example where s1 sends traffic with 3 priorities to s2 from `inport` 1 to `outport` 2, we can show how to reduce the memory usage of flow entries of s1 step by step. Our previous design [11], as shown in Figure 6(a), implements rate rescaling by replicating the flow entries with some parameter changes. This results in high memory usage as each flow entry includes duplicated information, such as `label` = 1 in each match part. We can reduce such duplication by using a greedy heuristic proposed in [22]. This still results in duplicated information between flow table 3 and flow table 4 as in Figure 6(b). Notice that these two tables are the same except the `outport`. We now propose a new method that can further reduce the duplicated information by the following analysis.

Notice that sometimes the values in `match` and `action` have correspondence between each. For instance, in Figure 6(a), ignoring others, for the same `inports` in the `match` field, the `outports` are also the same in the `action` field (`inport` = 1 corresponding to `outport` = 2 and vice versa). Therefore, we can further reduce the number of tables by recording such correspondence and checking it in each decomposition. In this example, finally we can get Figure 6(c) which saves one table compared to Figure 6(b) by adding the corresponding `outport` in the `action` of flow table 2.

This new flow table decomposition method is useful for Kuijia with multiple priorities. Traffic with different priorities goes through the same tunnels for each switch pair. If we built the flow tables based on simple replication as shown in Figure 6(a), most flow entries are almost the same except that different priority traffic needs to be tagged differently and matched to different priority queues. This yields a lot of opportunities for our decomposition method to exploit, and we can reduce the memory utilization of flow and group tables. For example, in Figure 7, the first flow entry in s1 is for traffic from s4 to s1. As there is only one outport for s1 in the simple topology, we only need to match the dst ip 10.0.0.0/21 of each packet in flow table 1 and then go to flow table 2 to check the `tc` (traffic class) filed in MPLS to decide which queue the packet should go to. Next it goes back to flow table 1 to execute next action `popmpls()` and output the packet from `outport` 4. The detailed explanation of the packet processing pipeline for multiple tables can be found in page 19 of [24].

Figure 8 shows the memory usage comparison between replicating flow entries and our new flow table decomposition method for the example shown in Figure 7. For simplicity, we treat the memory usage of each `match` (e.g., ipdst in ip range = 10.0.0.0/21) and `action` (e.g., `enqueue`(1)) the same—32 bits. Using our new method, the slope of the curve for multiple

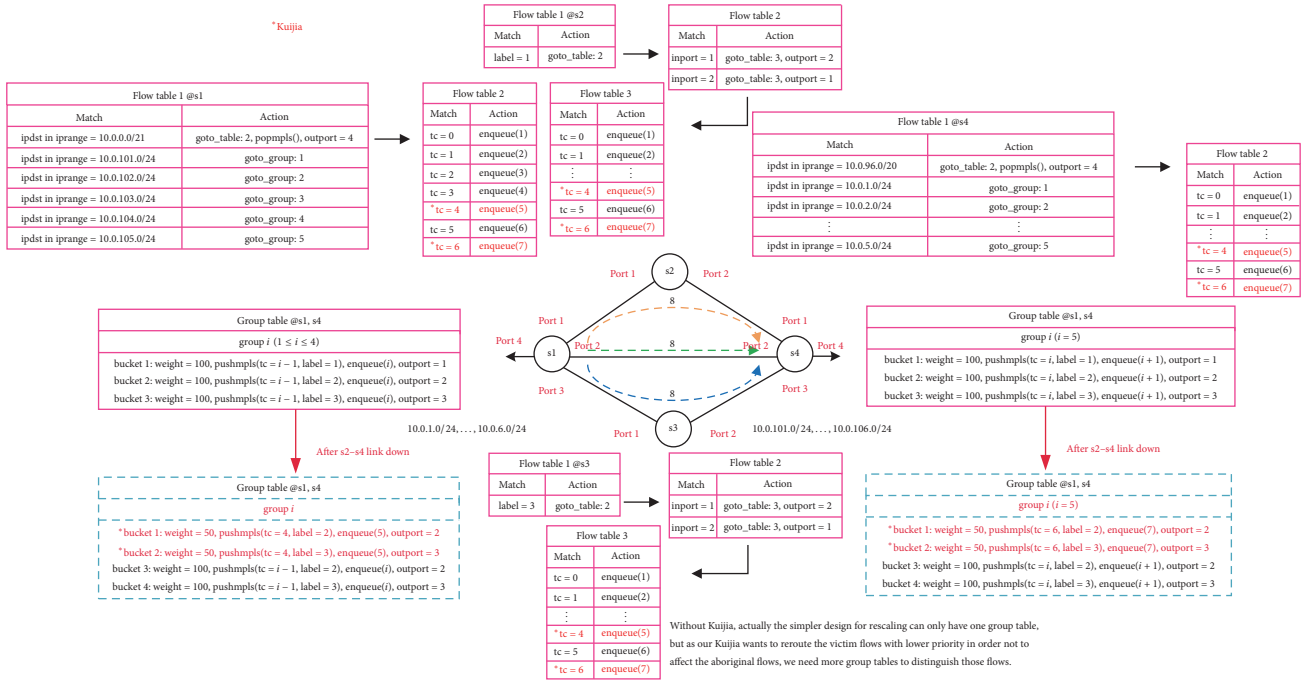


FIGURE 7: The design of flow table and group table of each switch in the simple topology under multiple priorities.

tables is about 0.36 times the slope of the curve for simple replicating method. This means our method can reduce more and more memory when the number of priorities increases (e.g., when there are 8 priorities, more than half of the memory of flow entries is saved).

5. Evaluation

We conduct comprehensive testbed experiments on Emulab to assess the effectiveness of Kuijia in this section. The evaluation details for nonpriority traffic are in the first three subsections and the fourth one shows the evaluation for multiple priority traffic.

5.1. Setup

Testbed Topology. We adopt a small-scale WAN topology for Google's inter-data center network reported in [1], which we refer to as the Gscale topology. There are 12 switches and 19 links as illustrated in Figure 9. We use a d430 node in Emulab running OVS to emulate a WAN switch in Gscale. Each link capacity is 1 Gbps. Each switch port has three queues: queue 0 is for control messages, queue 1 is for normal flows, and queue 2 is for rescaled flows. We test both TCP and UDP traffic sources using `iperf`.

TE Implementation. Similar to prior work [2, 6], we assume that there are 3 TE tunnels or paths between an ingress-egress switch pair. We use edge-disjoint paths whenever possible. The TE solution is obtained by solving a throughput maximization program using CVX. The corresponding group tables and flow tables are then configured by a RYU controller [25] at each switch. Rate limiting is done by the `Linux tc`.

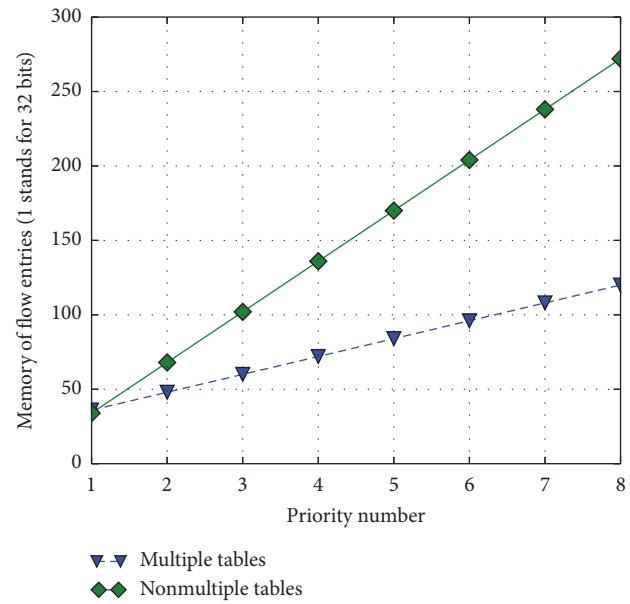


FIGURE 8: Memory utilization comparison of two designs for flow entries under multiple priorities.

Instead of generating a large number of individual flows between an ingress-egress switch pair, we simply launch 2 `iperf` aggregated flows on each TE tunnel and rescaling will reroute them to the two remaining tunnels separately after a single link failure. In total, there are 6 `iperf` aggregated flows for an ingress-egress switch pair. We determine the bandwidth of each `iperf` aggregated flow according to the weights of the tunnels. For example, if the TE result shows the bandwidth allocated to a switch pair is 300 Mbps and weights

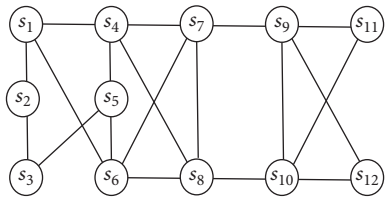


Figure 9: The Gscale topology.

for each tunnel are 0.5, 0.3, and 0.2, the bandwidth of the two `iperf` flows on the first tunnel is $300 * 0.5 * 0.3/(0.3 + 0.2) = 90$ Mbps and 60 Mbps, respectively. Similar to BwE [21], we use the DSCP field to carry the path ID in the packet header, since Emulab uses VLAN internally to connect its machines. We use the ECN bit as the priority tag. In environments when ECN or DSCP is already used, we can use other fields in IP options or MPLS for these purposes.

Now, since we do not have many flows, rescaling is implemented by a controller changing the action of the flow entries for the victim flows, so they are routed to the remaining tunnels. For Kuijia, the controller also changes the priority tag and sends the victim flows to the low priority queue after a failure.

Traffic. We use five random ingress-egress switch pairs in each experiment. We vary the demand of each switch pair from 0.8 Gbps to 1.6 Gbps in order to see Kuijia's performance with different extents of congestion. For each demand, we repeat the experiment three times and report the average.

5.2. Benefit of Kuijia. We first look at the benefit of Kuijia compared to rescaling. Three types of flows are affected by link failures. The first is the victim flows that are routed through the failed link. The second type is the directly affected flows, which are routed through path segments that the victim flows are rescaled to. The third type is the indirectly affected flows, which pass through path segments that the directly affected flows use. As these flows are hardly affected (less than 1% for rescaling and almost no effect for Kuijia in the experiments), we do not include them in the figures. Here, we focus on the directly affected flows. The results of victim flows are discussed in the next subsection.

For TCP flows, we evaluate the throughput loss after the failure for the directly affected flows shown in Figure 10. As the demand of each ingress-egress switch pair increases, the average throughput loss in terms of percentage for directly affected flows increases with the simple rescaling. This is because as demand increases, more links in the network may be fully utilized even before failure. After rescaling, they become congested and all flows passing these links suffer throughput loss. For Kuijia, as we reroute the victim flows with low priority, they are the only flows suffering packet loss and throughput degradation after failures. Thus, even when the demand is 1.6 Gbps for each ingress-egress switch pair, the average throughput loss of directly affected flows is little.

We also look at the convergence time of TCP after the link failure, which measures how long it takes for all flows to achieve stable throughput. Again, due to the cascading effect

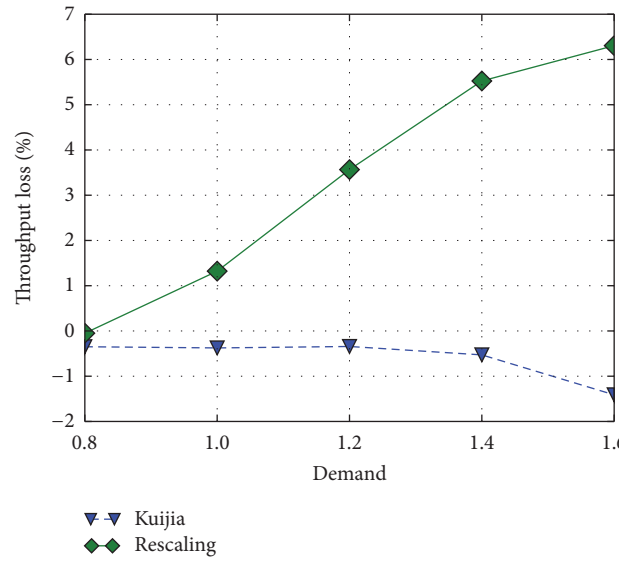


Figure 10: Throughput loss of directly affected TCP flows.

Table 4: Comparison of average TCP convergence time (s).

Link failure	Links 2-3 down	Links 7–9 down	Links 10-11 down
	Demand 0.8 Gbps		
Rescaling	1	1	1
Kuijia	<1	<1	<1
	Demand 1.0 Gbps		
Rescaling	3.75	4.50	1.75
Kuijia	<1	<1	<1
	Demand 1.2 Gbps		
Rescaling	11.00	12.00	12.75
Kuijia	<1	<1	<1
	Demand 1.4 Gbps		
Rescaling	10.50	16.00	12.25
Kuijia	<1	2.33	<1
	Demand 1.6 Gbps		
Rescaling	11.25	9.83	22.00
Kuijia	1.25	1.33	<1

of rescaling, all flows suffer from packet loss and enter the congestion avoidance phase. The convergence time is over 10 seconds when the demand exceeds link capacity as shown in Table 4. Now, with Kuijia, only victim flows need to back off, and thus the convergence time is greatly reduced to less than 1 second in almost all cases. One can also observe that the convergence time exhibits less variance with Kuijia compared to rescaling, since the congestion levels of tunnels can be vastly different with rescaling.

The benefit of Kuijia for UDP traffic is different. We use packet loss rate to measure the performance of UDP flows. The results are shown in Figure 11. For directly affected flows, packet loss rate with Kuijia is less than 2% in almost all cases, implying that the impact is negligible. Rescaling, on the other hand, results in much higher packet loss rates which are also increasing as demand increases.

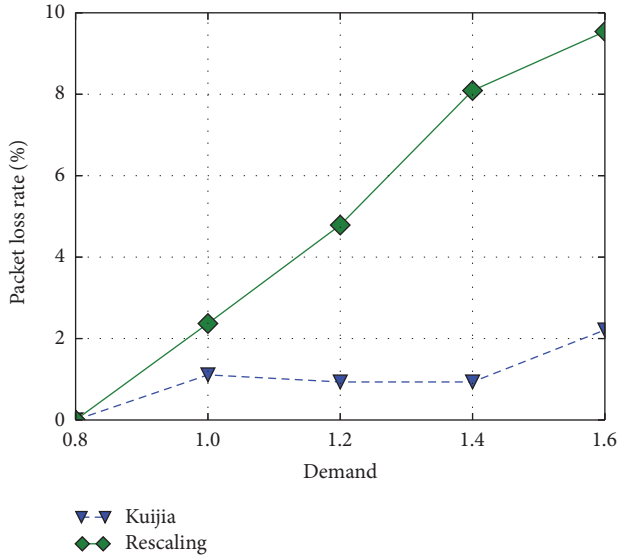


Figure 11: Packet loss rate of directly affected UDP flows.

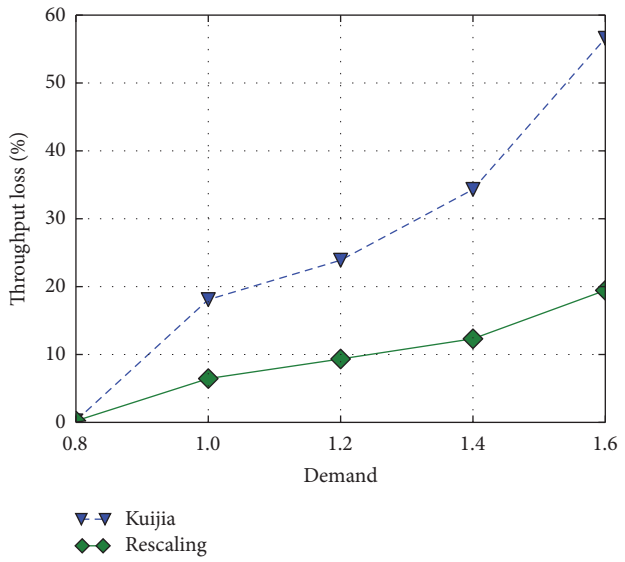


Figure 12: The overhead of TCP victim flows.

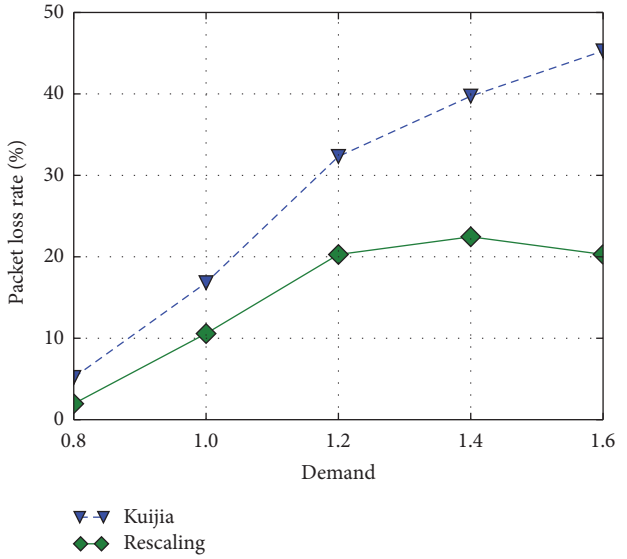


Figure 13: The overhead of UDP victim flows.

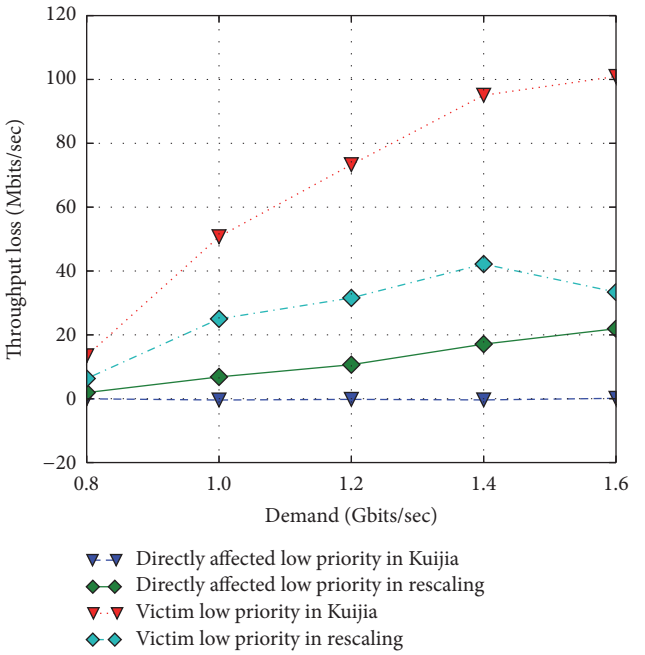


Figure 14: Throughput loss of low priority directly affected flows and victim flows under multiple priorities.

5.3. Overhead. Victim flows perform worse in Kuijia compared to rescaling, since they are the only flows that suffer throughput loss due to failures. We now look at this overhead of Kuijia. The result for both TCP and UDP traffic is shown in Figures 12 and 13. When demand of each switch pair increases, the average throughput loss of TCP victim flows and average packet loss rate of UDP flows also increase. We believe this is a reasonable trade-off to make, because in case of a link failure, traffic that traverses through this link is inevitably affected, especially when the demand exceeds link capacity in the first place. On the other hand rescaling causes too much collateral damage by making many other flows suffer from congestion, which should be avoided.

5.4. Benefit of Kuijia for Multipriority Traffic. We now use experiments to demonstrate the performance of Kuijia with multipriority traffic. Here, the setup is similar to Section 5.1. The difference is that now we have 4 queues for each port of the switches and each switch pair has 12 iperf TCP flows. Six flows are high priority traffic going through `queue(0)` before failure and `queue(1)` after failure if they are the victim flows. Correspondingly, the remaining six are for low priority traffic going through `queue(2)` before failure and `queue(3)` after failure if they are victim flows. Note that the setup is a simplification of the 8-priority design in Section 4.3. According to [1], we set the ratio of low/high priority traffic to 10.

The result is shown in Figure 14. As demand increases, the average throughput loss increases with simple rescaling

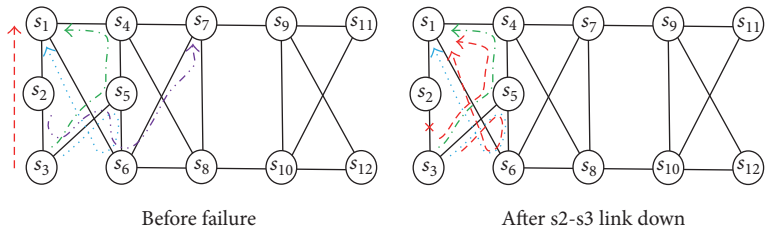


Figure 15: The partial flows' change after a link failure in one Gscale testbed experiment.

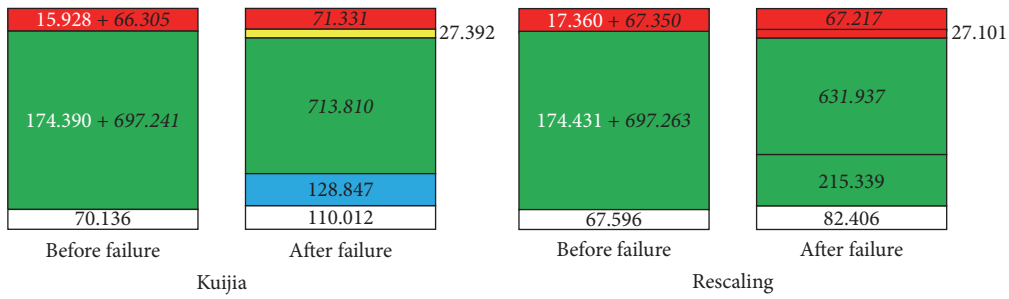


Figure 16: The traffic (Mbps) change of different priorities for flows passing s3 → s5 when demand = 1.2 Gbps.

for both victim low priority flows and directly affected low priority flows. For Kuijia, only the victim low priority flows suffer from throughput loss. The reason is similar to the explanation for Figure 10.

In order to compare the changes of traffic volume of different priorities after a link failure, we use the example as shown in Figure 15. Before failure, the purple (s2 → s3 → s5 → s6 → s7) and red (s3 → s2 → s1) flows pass through the link s2-s3 (two directions), and the green (s3 → s5 → s4 → s1) and blue (s3 → s5 → s6 → s1) flows pass through the link s3 → s5 (single direction). After the link s2-s3 link is down, the purple flows are rerouted to s7 without taking s3 → s5, so we remove it in the right topology of Figure 15. The red flows are rerouted through the remaining tunnels.

We compute the traffic volume of flows passing through the link s3 → s5 before and after failure for each priority and show the comparison result in Figure 16. The same color represents the same priority traffic, and from up to down, the priority decreases. The white area is traffic volume of the purple flows which will be rerouted away after s2-s3 link is down. As the ratio of low/high priority traffic is 10, we can observe that, for both Kuijia and rescaling, the high priority directly affected flows (the italic numbers in the top red area) and high priority victim flows (the black numbers on the right side of the yellow area for Kuijia and of the second red area for rescaling) are almost not affected by the link failure. However, for rescaling, the low priority directly affected flows suffer the throughput loss (697.263 Mbps decreases to 631.937 Mbps) as it reroutes the victim flows with unchanged priorities. When the number of priorities increases and the volume of rerouted victim flows increases, the benefit of Kuijia can be more salient.

To evaluate the benefit of our new flow table decomposition method proposed in Section 4.3, we count the number of `match` fields and `actions` of each flow entry in the experiment (e.g., a flow entry: `match` {tc = 0, inport = 1, label = 1}, `actions` {enqueue(1), outport = 2}; the number of fields is 3 + 2 = 5). We find that our new flow table decomposition method can reduce the average number of fields for all the flow entries used in the experiment at least from 2552 (without using multiple tables) to 1580. It can save about 38% memory.

6. Discussion

We now discuss several issues pertaining to the use of Kuijia in a production data center WAN.

Traffic Characteristics. The benefit of Kuijia depends on the traffic's characteristics. For elastic traffic like file transfer which is TCP friendly, no matter how large the bandwidth is, the file can be received finally. Therefore, in Google BwE [21], many of their WAN links run at 90% utilization by sending elastic traffic at a low priority. In these cases, rescaling may be good enough as it shares the bandwidth after rescaling across many users. With Kuijia, some users have to suffer significant throughput loss which may stall their transmission and hurt their experience.

However, for inelastic traffic like video conferencing, video streaming, online search, or stock trading, they all need a certain level of bandwidth to be delivered on time with quality. As a result, Kuijia is much better here as it limits the impact of failures to the victim flows. It is much worse when the cascading effect of simple rescaling causes many users to experience playback delay, whereas Kuijia limits the performance impact to only the victim flows that have to suffer from failures no matter what. With the rapid growth of mobile traffic, there will be more and more inelastic traffic, creating more use cases for Kuijia in data center WANs.

Traffic Priorities. As a data center WAN carries both elastic and inelastic traffic, it usually employs priorities to differentiate the QoS [1, 2, 21]. Inelastic traffic is given high priority while elastic traffic is given low priority [1, 2, 21]. Thus, it is important to consider multipriority when dealing with failures. In such a case, Kuijia's potential benefit can be more significant in the future, given the growth of high priority inelastic traffic such as video.

Impact of Flow Size to Hashing. In intra-data center networks, it is known that hashing based ECMP leads to suboptimal performance due to flow size imbalance. A few elephant flows may be hashed to the same path among many choices creating hotspots in the network. This does not happen in data center WANs. The WAN carries aggregated flows over more than thousands of individual TCP flows across the wide area, using a few sets of tunnels [1, 2, 6, 21]. The aggregated behavior of flows is a persistent flow with infinite data to send, which is the common abstraction used in the literature [1, 2, 6, 21]. TE calculates the splitting ratios of the aggregated flow across a few tunnels as well as the sending rate for the next interval. Hashing works well and can achieve the splitting ratios given by TE when the actual number of TCP flows is extremely large compared to the number of paths (tunnels) available. Thus, hash imbalance is not an issue.

7. Conclusion

We develop Kuijia, a robust TE system for data center WANs based on rate rescaling method, to reduce the affected flows due to data plane faults by rerouting the victim flows from failure tunnels to other healthy tunnels with lower priority. This protects the aboriginal traffic of those healthy tunnels from congestion and packet loss, as the traffic from the failure tunnels will suffer them. By evaluating our method in Emulab Gscale testbed that we implemented, the results show that Kuijia works well for both nonpriority traffic and multipriority traffic whether in pure SDN network or in a hybrid network like Emulab.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work is supported in part by the Hong Kong RGC ECS-21201714, GRF-11202315, and CRF-C7036-15G.

References

[1] S. Jain, A. Kumar, S. Mandal et al., "B4: Experience with a globally-deployed software defined WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2013*, pp. 3–14, China, August 2013.

[2] C.-Y. Hong, S. Kandula, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proceedings of the ACM Conference on SIGCOMM*, pp. 15–26, ACM, Hong Kong, August 2013.

[3] M. Zhang and H. H. Liu, *Private Conversation with The Authors, Microsoft Research*, March 2015.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM*, pp. 254–265, New York, NY, USA, August 2011.

[5] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2013*, pp. 411–422, China, August 2013.

[6] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2014*, pp. 527–538, USA, August 2014.

[7] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proceedings of the 2015 International Conference on Computing, Networking and Communications, ICNC 2015*, pp. 77–81, USA, February 2015.

[8] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 1086–1097, 2015.

[9] Q. Yan and F. R. Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 52–59, 2015.

[10] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," in *Proceedings of the the ACM SIGMETRICS joint international conference*, p. 97, San Jose, California, USA, June 2011.

[11] C. Zhang, H. Xu, L. Liu et al., "Kuijia: Traffic rescaling in data center WANs," in *Proceedings of the 37th IEEE Sarnoff Symposium, Sarnoff 2016*, pp. 142–147, USA, September 2016.

[12] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, pp. 109–114, China, August 2013.

[13] S. H. Yeganeh and Y. Ganjali, "Beehive: Towards a simple abstraction for scalable software-defined networking," in *Proceedings of the 13th ACM SIGCOMM Workshop on Hot Topics in Networks, HotNets 2014*, USA, October 2014.

[14] L. Schiff, M. Borokhovich, and S. Schmid, "Reclaiming the brain: Useful OpenFlow functions in the data plane," in *Proceedings of the 13th ACM SIGCOMM Workshop on Hot Topics in Networks, HotNets 2014*, USA, October 2014.

[15] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing OpenFlow graph algorithms," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 121–126, USA, August 2014.

[16] Y. Chang, S. Rao, and M. Tawarmalani, "Robust validation of network designs under uncertain demands and failures," in *Proceedings of the USENIX NSDI*, 2017.

[17] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "We've got you covered: Failure recovery with backup tunnels in traffic

engineering," in *Proceedings of the 24th IEEE International Conference on Network Protocols, ICNP 2016*, Singapore, November 2016.

[18] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Rule-Caching Algorithms for Software-Defined Networks," Tech. Rep., Princeton University, 2014.

[19] The University of Utah, Emulab, 2017, http://www.emulab.net/.

[20] B. Pfaff, J. Pettit, T. Koponen et al., "The design and implementation of open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, pp. 117–130, usa, May 2015.

[21] A. Kumar, S. Jain, U. Naik et al., "BwE: flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 1–14, London, UK, August 2015.

[22] L. Molnár, G. Pongrácz, G. Enyedi et al., "Dataplane specialization for high-performance OpenFlow software switching," in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2016*, pp. 539–552, Brazil, August 2016.

[23] J. M. Wang, Y. Wang, X. Dai, and B. Bensaou, "SDN-based multi-class QoS-guaranteed inter-data center traffic management," in *Proceedings of the 2014 3rd IEEE International Conference on Cloud Networking, CloudNet 2014*, pp. 401–406, Luxembourg, October 2014.

[24] Open Networking Foundation, OpenFlow Switch Specification 1.5.1, 2015, https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf.

[25] SDN Framework Community, RYU, 2016, https://github.com/osrg/ryu.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 7545079, 16 pages
https://doi.org/10.1155/2018/7545079

Research Article

SDNManager: A Safeguard Architecture for SDN DoS Attacks Based on Bandwidth Prediction

Tao Wang, Hongchang Chen, Guozhen Cheng, and Yulin Lu

National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450002, China

Correspondence should be addressed to Tao Wang; wangtaogenuine@163.com

Received 28 September 2017; Revised 24 November 2017; Accepted 11 December 2017; Published 15 January 2018

Academic Editor: Qiang Fu

Software-Defined Networking (SDN) has quickly emerged as a promising technology for future networks and gained much attention. However, the centralized nature of SDN makes the system vulnerable to denial-of-services (DoS) attacks, especially for the currently widely deployed multicontroller system. Due to DoS attacks, SDN multicontroller model may additionally face the risk of the cascading failures of controllers. In this paper, we propose SDNManager, a lightweight and fast denial-of-service detection and mitigation system for SDN. It has five components: monitor, forecast engine, checker, updater, and storage service. It typically follows a control loop of reading flow statistics, forecasting flow bandwidth changes based on the statistics, and accordingly updating the network. It is worth noting that the forecast engine employs a novel dynamic time-series (DTS) model which greatly improves bandwidth prediction accuracy. What is more, to further optimize the defense effect, we also propose a controller dynamic scheduling strategy to ensure the global network state optimization and improve the defense efficiency. We evaluate SDNManager through a prototype implementation tested in a real SDN network environment. The results show that SDNManager is effective with adding only a minor overhead into the entire SDN/OpenFlow infrastructure.

1. Introduction

Software-Defined Networking (SDN) [1] has quickly emerged as a new paradigm that decouples the control logic from data plane devices. It offloads the complex network control functions to the logically centralized controllers, while the data plane tends to be a set of dumb forwarding devices. Controllers, as the main component of SDN, are responsible for maintaining network-wide state views and performing forwarding decisions to support fine-grained network management policies. Therefore, the separation of control plane and data plane enables a flexible network management and rapid deployment of new functionalities. However, security of controller is the precondition and the basis of SDN.

OpenFlow as a reference implementation of SDN has been widely used in recent years. In OpenFlow, when a switch receives a new flow and there are no matching flow rules installed in its flow table, the data plane will typically buffer the packet in the first place and then request a new flow rule from the controller with an OFPT PACKET IN message. However, it is obvious to see that the centralized controller introduces considerable overhead and could easily become a bottleneck. As illustrated in Figure 1, two kinds of DoS attacks [2] are generally considered in SDN networks. In the first attack, attackers may simply mount saturation attacks towards an SDN controller by sending massive useless packets. Then the controller has to handle every useless new flow for flow entry creation, which greatly occupies computing resource and makes controllers show poor responsiveness to other legitimate flow requests. In the second attack, due to the limited storage capacity, the switch can only store a certain amount of flow entries. So if the attacker aims to saturate the memory, the switch cannot insert the new flow entry into switch flow table and will respond with an error message to the controller, simply dropping the packets at the end.

Existed control plane is typically equipped with one controller, which makes SDN more vulnerable to DoS attack. For example, if the only controller is overwhelmed or compromised by attacks, the entire network system will be paralyzed subsequently. Thus, to improve scalability and alleviate single point failure, the latest control plane is usually implemented as a distributed network operating system with

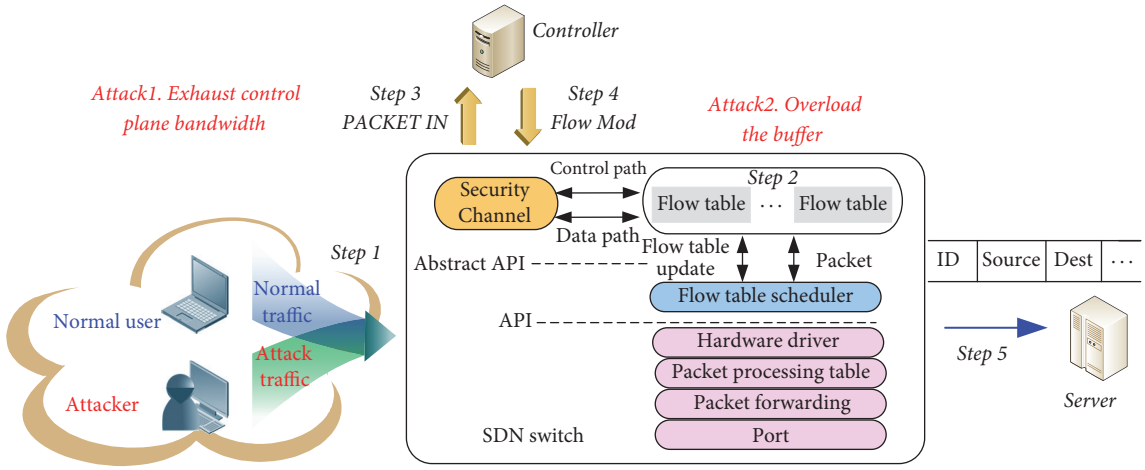


Figure 1: SDN architecture and denial-of-service attacks.

multiple controllers (ONOS). Wang et al. [3] dynamically assign controllers to minimize the average response time and balance load. ElDefrawy and Kaczmarek [4] introduce a prototype SDN controller that can tolerate Byzantine faults. Note that current multiple-controller architectures are mostly designed from the aspects of scalability and efficiency instead of typical security, so a more comprehensive architecture which emphasizes on the aspect of security urgently needs to be solved. What is more, in the multiple-controller model, if one controller fails after a successful DoS attack, other controllers will take over it, and the additional load may make them overloaded, causing the cascading failure [5] ultimately. As illustrated in Figure 2 *(Stage 1)*, when controller c fails, the load of controller c (switches 7, 8, and 9) is redistributed to controller a and d, whose loads are then exceeding their capacities. In *Stage 2*, when controllers a and d fail, the load of controller a (switches 1, 2, 3, 4, 5, 6, and 7) is redistributed to controller b, and the load of controller d (Switches 8, 9, 15, 16, 17, and 18) is also redistributed to controller b too. In *Stage 3*, controller b takes all the load to manage the network. In *Stage 4*, controller d fails at last, and all the switches are out of control. As a result, the whole SDN network becomes paralyzed and triggered by the failure of only one controller, which accelerates the paralysis of the whole model.

To address the above problems, we devised SDNManager, a fast and lightweight denial-of-service detection and mitigation system for SDN, which can significantly increase the resistance of SDN networks to both single point of failure and cascading failure caused by DoS attack. Specifically, our contributions are as follows:

(i) SDNManager employs a novel dynamic time-series (DTS) model which greatly improves bandwidth prediction accuracy, so it can provide higher detection accuracy and ensure better protection to the whole networks.

(ii) SDNManager is implemented on the control plane, which conforms to the SDN security trend and does not require any modification on the data plane. Therefore, it is easier to deploy SDNManager compared to the previous solutions.

(iii) SDNManager is valid for all types of DoS attacks.

(iv) SDNManager adds minor overhead into the entire SDN/OpenFlow infrastructure.

(v) To further optimize the defense effect, we also propose a controller dynamic scheduling strategy to ensure the global network state optimization and improve the defense efficiency.

The rest of the paper is organized as follows. Section 2 introduces some related works. The design of SDNManager is detailed in Section 3. Section 4 presents the dynamic controller scheduling strategy. The experiments and results of the SDNManager and dynamic controller scheduling strategy evaluation are presented in Sections 5 and 6, respectively. Section 8 introduces our future works. Section 8 concludes the paper.

2. Related Work

Some recent research [6–8] also pointed out that the SDN controller is a vulnerable target of DDoS attacks. Yan and Yu [9] argue that although SDN controller itself is a vulnerable target of DDoS attacks, it also brings us an unexpected opportunity to mitigate DDoS attacks in cloud computing environments. Several solutions have been proposed to mitigate the SDN DoS attacks. For example, Kotani and Okabe [10] proposed a packet-in message filtering mechanism for protection of the SDN control plane. The packet-in filtering mechanism can first record the values of packet header fields before sending packet-in messages and then filter out packets that have the same values as the recorded ones. Of course, if the DoS attacker deliberately sends new packets that have different values from the recorded ones, the packet-in filtering mechanism will be completely ineffective. Mousavi and St-Hilaire [11] introduced an early detection method for DDoS attacks against the SDN controller based on the entropy

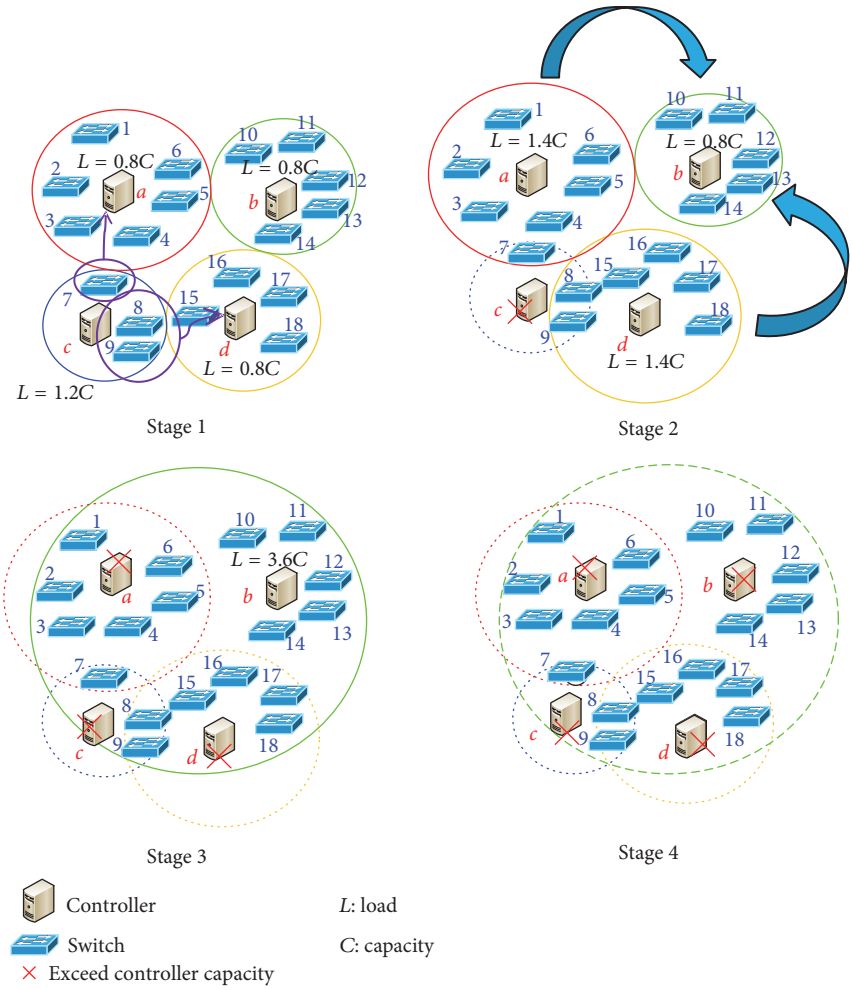


Figure 2: The cascading failures of controllers.

variation of destination IP address. He thinks that the destination IP addresses of normal flows should be almost evenly distributed, while the destination IP addresses of malicious flows are always destined for several targets. However, it is also not difficult for DoS attackers to generate a large amount of new traffic flows, the destination IP addresses of which are evenly distributed, to overload the SDN controller. AvantGuard [2] introduces connection migration and actuating triggers into the SDN architecture to defend against the SYN Flood attacks, but it does not work well when confronted with other DoS attacks in SDN. FloodGuard [12] uses proactive flow rule analyzer and packet migration to defend against data plane saturation attack, but it is too costly. Previously related works, such as [13, 14], employed Self Organizing Maps (SOM) to classify whether the traffic is abnormal or not to defend against DoS attacks, but the overhead of the classification is also too high to be used in real time. Yan et al. [15] proposed a solution to detect DDoS attacks based on fuzzy synthetic evaluation decision-making model. Although it is a lightweight detection method, its detection accuracy is not very well. Wang et al. [16] formulate the dynamic controller assignment problem as an online optimization problem aiming at minimizing the total cost. They also propose a novel two-phase algorithm by casting the assignment problem as a stable matching problem with transfers. Simulations show that the online approach reduces total cost and achieves better load balancing among controllers. However, the algorithm is mostly devised from the perspective of scalability, efficiency, and availability instead of security.

Wei et al. [17] employ the ARIMA model and the GARCH model to forecast the trend and volatility of the future demand. Amin et al. [18] propose a forecasting approach that integrates ARIMA and GARCH models to capture the QoS attributes' volatility. Krithikaivasan et al. [19] develop a forecasting methodology based on an ARCH model which captures the time variation in the (conditional) variance of a time-series. However, the above models have shown that the statistical distribution of the innovations (errors) often has a departure from normality which is typical of the volatility found in bursty traffic. Furthermore, the past linear combinations of squared error terms have been found to lead to inaccurate forecasts. Therefore, we employ the adaptive conditional score of the distribution to track volatility in DTS model.

By analyzing the existing works, it is not difficult to find that a more systematic approach needs to be designed to deal with the attacks mentioned above. Next, we will show our design.

3. Design of SDNManager

In the current OpenFlow reactive mode, network suffers from DoS attacks due to a lack of flow-level bandwidth control. In other words, attackers can send useless packets without limits, and then controller must perform forwarding decisions and generate flow rules unconditionally until the controller is saturated. Furthermore, current SDN designs widely utilized multiple controllers. Besides exploiting heterogeneity and dynamism to improve the security level of NOS in some other aspects, it also increases vulnerability in the single point of failure and easily causes the cascading failures of controllers, which makes the whole network be paralyzed more quickly. Thus, it is undoubted that the influence is significant, once controllers are flooded.

In this paper, we introduce SDNManager, a fast and lightweight denial-of-service detection and mitigation system that allows multiple controllers to operate independently to mitigate denial-of-service attack on the controller. SDNManager typically follows a control loop of reading flow statistics, forecasting flow bandwidth changes based on the statistics and accordingly updating the network. Flows with bandwidth usage higher than the predicted bandwidth usage are penalized by the application. The penalization is proportional to the difference between current usage and predicted usage. Therefore, the service will not be disrupted, and the cascading failures can be effectively avoided even though some controllers are under DoS attacks. Figure 3 shows the architecture of SDNManager. The details of SDNManager are described in the rest of this subsection. Notations of SDNManager lists most of the notations used in the paper.

3.1. System Architecture. As Figure 3 shows SDNManager mainly has five components: monitor, forecast engine, checker, updater, and storage service. We outline the role of each component by discussing three kinds of operating states of SDNManager.

In *observed state* (OS), monitor periodically collects an up-to-date view of the actual network states, including flow rules statistics and path conditions, from switches, and transforms them into OS variables. Then, the forecast engine reads these variables and forecasts the expected bandwidth utilization for each flow based on its historical data trace. Using a bandwidth checker module, SDNManager merges these *proposed states* (PS, the expected bandwidth utilization) into a *target state* (TS) that is guaranteed to maintain the safety and performance of the system. Updater module then updates the network to the target state. What is more, storage service as the center of the system persistently stores the variables of OS, PS, and TS and offers a highly available, read-write interface for other components, which greatly simplifies the design of the other components and allows

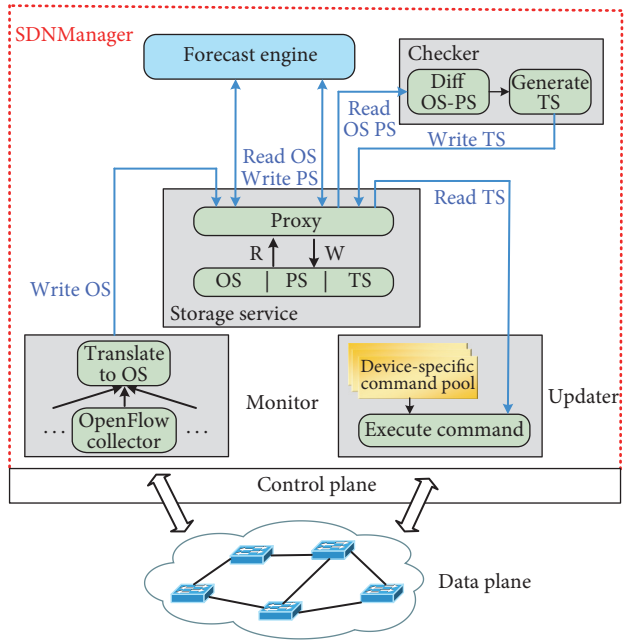


Figure 3: The architecture of SDNManager.

them to be stateless. Of course, what counts above all is that SDNManager is supposed to forecast bandwidth utilization accurately. Thus, we employ a dynamic time-series model to develop a novel bandwidth utilization forecast engine. Next, we introduce the forecast engine in detail.

3.2. Forecast Engine. An ARCH (autoregressive conditionally heteroscedastic) model is a model for the variance of a time-series. ARCH models are used to describe a changing, possibly volatile variance. Current research in forecasting volatility adopted methods developed originally from the ARCH model in Econometrics. Although an ARCH model could be used to describe a gradually increasing variance over time, most often it is used in situations in which there may be short periods of increased variation. Also, it has been observed that the statistical distribution of the innovations (errors) often has a departure from normality which is typical of the volatility found in bursty traffic. Furthermore, the past linear combinations of squared error terms have been found to lead to inaccurate forecasts. Therefore, ARCH model is not our best choice, and we need to optimize it. Inspired by the ARCH model [22], we adopt a dynamic time-series model (DTS) to forecast bandwidth volatility. More specifically, we employ the adaptive conditional score of the distribution to track volatility in DTS model. Of course, before describing the DTS model in detail, we first briefly introduce the ARCH model. The ARCH model illustrates the variance of a time-series as a function of the squares of its past observations. The major contribution of the ARCH model is to find that apparent changes in the volatility of bandwidth time-series may be predictable. An ARCH process indexed on time t as a random variable Y_t of order ($m > 1$) can be expressed as

(1) In each time slot t
(2) **For** each controller $C_i \in C$
(3) *Query* flow statistics and Path conditions
(4) $\mathrm{OS}_{C_i} \leftarrow$ Transform(state variables)
(5) **for** each flow $\mathrm{flow}_{ij} \in \mathrm{flow}_{C_i}$
(6) $\mathrm{PS}_{C_i} \leftarrow \mathrm{BW}_{\mathrm{flow}_{ij}.\mathrm{fcst}} = \mathrm{ForecastingEngine}(\mathrm{OS}_{C_i})$
(7) **If** $\mathrm{BW}_{\mathrm{flow}_{ij}}/\mathrm{BW}_{\mathrm{flow}_{ij}.\mathrm{fcst}} > 1 + \xi$
(8) $\mathrm{TS}_{C_i} \longleftarrow \mathrm{BW}_{\mathrm{flow}_{ij}.\mathrm{target}} = \mathrm{BW}_{\mathrm{flow}_{ij}} * \mathrm{pun}\left(\frac{1}{e^{\mathrm{BW}_{\mathrm{flow}_{ij}} - \mathrm{BW}_{\mathrm{flow}_{ij}.\mathrm{fcst}}}}\right)$
(9) **else**
(10) $\mathrm{TS}_{C_i} \leftarrow \mathrm{BW}_{\mathrm{flow}_{ij}.\mathrm{target}} = \mathrm{BW}_{\mathrm{flow}_{ij}}$
(11) **end if**
(12) **end for**
(13) **End For**
(14) *Enforce* Load redistribution strategy
(15) **Return**

Algorithm 1: The main procedure of SDNManager.

a product of innovations (errors) of the past m terms and its standard deviation by

$$Y_t = \sigma_t \varepsilon_t,$$
$$\sigma_t^2 = \omega + \sum_{i=1}^{m} a_i Y_{t-i}^2 + \sum_{j=1}^{m} b_j \sigma_{t-j}^2. \tag{1}$$

Here, $\{\omega, a_i, b_j\} > 0$ are constants obtained through the process of model fitting. However, considering the above equations, we can observe that the statistical distribution of the ARCH model always has the departure from normality which is typical of the volatility existed in highly dynamic SDN traffic. Furthermore, past linear combinations of squared error terms can inevitably bring about inaccurate forecasts. Thus, we employ the conditional score of the distribution to track volatility to solve these problems.

We also define a random variable BW_t as the throughput of the time-series elicited from analyzed traces, whose t_{th} observation is the dependent variable of interest with a time-varying parameter f_t which will be determined by estimating θ_t, the parameter of the conditional distribution of BW_t.

$$\mathrm{BW}_t \sim p\left(\mathrm{BW}_t \mid f_t; \theta_t\right). \tag{2}$$

Here, we employ Maximum Likelihood Estimation (MLE) where the parameter θ_t is determined by the population that most likely produced the data vector BW_t. The following equation is a recursion expression for f_t:

$$f_t = \omega + \sum_{i=1}^{n} \alpha_i f_{t-1} + \sum_{j=1}^{m} \beta_j s_{t-1}. \tag{3}$$

Here, ω is a constant and $\{\alpha_i, \beta_j\}$ are coefficient matrices obtained through the process of model fitting and dimensioned for $\{i, j = 1 : n; 1 : m\}$, respectively. When an observation density is realized for BW_t, the time-varying parameter f_t is updated by the conditional score s_t:

$$s_t = s_t \nabla_t, \tag{4}$$

where $\nabla_t = \partial \log p(\mathrm{BW}_t \mid f_t; \theta_t)/\partial f_t$ is the first derivative of the log-likelihood function of the conditional distribution's parameter. Inserting back into (3) yields

$$f_{t+1} = \omega + \alpha f_t + \beta s_t \left[\frac{\partial \log p\left(\mathrm{BW}_t \mid f_t; \theta_t\right)}{\partial f_t}\right]. \tag{5}$$

The variance will be the second derivative in keeping with a measure of volatility. The second moment is given by the following equation:

$$I_t = -E\left[\frac{\partial^2 \log p\left(\mathrm{BW}_t \mid f_t; \theta_t\right)}{\partial^2 f_t}\right] = E\left[\nabla_t \nabla_t'\right]. \tag{6}$$

Here, I_t is the Fisher information matrix for all terms to be used in computing the volatility and ∇_t is the score vector. Therefore, the model takes historical data trace as an input and makes full use of the adaptive conditional score to yield the predicted flow-level bandwidth utilization BW_t. The DTS model makes full use of the observational density of the dependent variable in updating the conditional score rather than a linear combination of a finite number of past variance terms. In addition, it also employs the derivative as a better filter with the conditional density and is, therefore, less prone when outliers are present in historical data.

3.3. Dynamic Bandwidth Allocation Algorithm. After completing the design of the forecast engine, we can then develop an integrated dynamic bandwidth allocation algorithm for this issue. The pseudocode of the dynamic bandwidth allocation algorithm is shown in Algorithm 1. The practical solution uses explicit bandwidth information of each flow to enforce accurate rate control for traffic flows between controllers and switches. The solution takes full advantage of the ability of global monitoring in SDN. The explicit steps of the algorithm are as follows.

First, the monitor periodically collects the current flow statistics and path conditions and transforms them into OS

variables (lines: (3)-(4)). Second, the forecast engine reads the latest OS variables from the storage service and proposes the expected bandwidth utilization for each flow (line: (6)). Then, bandwidth checker examines whether the observed bandwidth utilization for each flow is consistent with the expected bandwidth utilization (line: (7)). We set the slack variable ξ [23], mainly to reduce the impact of SDNManager on normal flow. Flows with bandwidth consumption higher than the predicted bandwidth consumption will be penalized by the application (bandwidth checker). The penalization is proportional to the difference between current and predicted usage (lines: (8)–(10)). Finally, each controller repeats the above steps and takes load redistribution strategy to prevent cascading failures (line: (14)). In addition, there is a problem worthy of attention, once the predicted bandwidth is not accurate and the flow is penalized for that mistake, then the user of the flow can be hurt, which is not fair. Considering the predicted bandwidth is not absolutely accurate, we set the slack variable $\xi \in [0, 1]$ in Algorithm 1, mainly to reduce the impact of SDNManager on normal flow. Of course, it is necessary to ensure both the "fairness" and "security" of SDNMManager. Here "fairness" means that even if the predicted bandwidth is not absolutely accurate, SDNManager should guarantee the normal flow will not be penalized for the mistake. "Security" of the process means that SDNManager should prevent possible DoS attacks. Therefore, we can balance the fairness and security by adjusting the value of the slack variable based on our actual requirements. For example, if we pay more attention to "fairness," the value of ξ could be set to big enough. On the contrary, if we pay more attention to "security," the value of ξ could be set to small enough. Compared with the previous rate limiting algorithm [24], our method is more moderate. The previous rate limiting algorithm is not fair because it penalizes all routers equally, irrespective of whether they are greedy or well behaving.

4. Dynamic Controller Scheduling Strategy (DCS)

With SDN technology widely used in the cloud data center [25], the industry uses SDN multicontroller model [26, 27] to achieve high performance and high scalability. However, the SDN multicontroller model is more vulnerable to the DoS attacks on the controller. Therefore, SDN multicontroller model does need the associated SDN DoS defense mechanism to enhance its availability. The defense mechanism designed in this paper is adaptive for the SDN multicontroller model. It can effectively prevent the DoS attack against the controller and avoid the single failure point of the controller [28]. Of course, the defense mechanism also inevitably increases the controller load. Also, given the random nature of DoS attacks (random time, random address, random attack rate, and random controller), it can result in unbalanced load across multiple controllers, affect the average response time of the global network, and reduce the overall quality of service. Therefore, to further optimize the defense effect, we propose a controller dynamic scheduling strategy to ensure the global network state optimization and improve the defense efficiency.

4.1. Dynamic Controller Scheduling Model Establishment. We assume that the SDN network consists of M controllers and N switches. The controller set and the switch set are represented by $C = \{c_1, c_2, \ldots, c_m\}$ and $S = \{s_1, s_2, \ldots, s_n\}$, respectively. c_m and s_n represent the m_{th} controller and the n_{th} switch, respectively. $y(t)_{ij}$ indicates whether the i_{th} switch is connected to the j_{th} controller (1 indicates that the connection is established, 0 indicates that the connection is not established). d_{ij} is the distance between the i_{th} switch and the j_{th} controller (hops). The processing capability of each controller in the controller set C is $\mu = \{\mu_1, \mu_2, \ldots, \mu_m\}$. In order to increase the controller reliability and elasticity, we set the decay factor of each controller as $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$, where $\gamma \in (0, 1)$.

4.1.1. Average Global Controller Response Time. Switch requests are aggregated at the processing queue of the connected controllers. Therefore, if the i_{th} switch sends requests to the controller at the rate $\upsilon(t)_i$ in time slot t, the load of controller j can be expressed as

$$\theta(t)_j = \sum_{i=1}^{N} \upsilon(t)_i \, y(t)_{ij}, \tag{7}$$

where $y(t)_{ij}$ indicates whether the i_{th} switch is connected to the j_{th} controller (1 indicates that the connection is established, 0 indicates that the connection is not established). According to the (7) and taking into account the effect of the network topology size on the average response time of the controller, we use (8) to compute the average request response time for controller j:

$$\vartheta(t)_j = \frac{1}{\mu_j - \theta(t)_j} O\left(V^2\right), \tag{8}$$

where V is the scale of network topology (i.e., the number of network nodes).

The average controller response time in time slot t can be represented as the weighted average of $\{\vartheta(t)_j\}$. Therefore, according to (7) and (8), the average global controller response time is

$$\xi(t) = \frac{\sum_{j=1}^{M} \theta(t)_j \, \vartheta(t)_j}{\sum_{j=1}^{M} \theta(t)_j}. \tag{9}$$

4.1.2. Dynamic Controller Scheduling Strategy. Given the SDN multicontroller model, the dynamic controller scheduling strategy can dynamically assign controllers to switches according to the change of the controller load. By adjusting the mappings between controllers and switches, we can reduce the average response time of the global controller and optimize the global quality of service. The model can be abstracted as follows:

$$\text{Minimize} \quad \xi(t) \tag{10}$$

$$\text{s.t.} \quad \theta(t)_j \le \gamma_j \mu_j, \quad \forall j \tag{11}$$

Input: $\forall i, \upsilon(t)_i; \forall j, \mu_j, \gamma_j$
Output: $y(t)_{ij}, \forall i, j$
(1) **function** BSM $(\upsilon(t)_i, \mu_j, \gamma_j)$
(2) Each switch and controller builds $\Gamma(s_i)$ and $\Gamma(c_j)$
(3) **while** Any proposals from the switches **do**
(4) **if** All the proposals will not vilolate constraint (11) **then**
(5) The controllers accept all the proposals
(6) **else**
(7) The controllers only accept the most preferred proposals based on $\Gamma(s_i)$
(8) The controllers reject other proposals
(9) **end if**
(10) **end while**
(11) Transform matching Θ to $y(t)_{ij}, \forall i, j$
(12) **end function**

Algorithm 2: Bidirectional selection mechanism.

$$\sum_{j=1}^{M} y(t)_{ij} = 1, \quad \forall i \tag{12}$$

$$y(t)_{ij} \in \{0, 1\}, \quad \forall i, j. \tag{13}$$

Formula (10) ensures the average response time of the global controller is optimal. Constraint (11) ensures that no controller is overloaded. Constraint (12) ensures that each switch must be connected to only one controller so as to ensure validity and uniqueness.

4.2. Solution for Dynamic Controller Scheduling Model. Because the controller dynamic scheduling strategy is oriented to the SDN multicontroller model, its solution needs to satisfy the high efficiency and the rapid convergence of the dynamic environment. Through solving the model, we can get the global optimal controller-to-switch mapping. The optimal mapping will balance the load across the multiple SDN controllers and optimize the defense effects of the mechanism.

In order to solve the optimal dynamic controller scheduling model, we first make the controller and the switch perform "bidirectional selection" to construct the initial controller-to-switch mapping and then use the iterative algorithm to adjust the mappings to achieve global network performance optimization. Next, we will describe the solution for dynamic controller scheduling model in detail in the following subsections.

4.2.1. "Bidirectional Selection" Mechanism. In this paper, the "bidirectional selection" mechanism is used to construct the initial controller-to-switch mapping. More specifically, all controllers and switches maintain their preferences list based on some metrics. In the case of satisfying the global constraints, we construct the initial mapping according to the preference lists [29].

From the perspective of the switch, it may choose the controller with higher processing power and shorter response time (such as formula (8)) in the first place. However, this indicator is related to many other factors, so it is not easy to make a choice. Therefore, we use the maximum controller response time as the indicator to construct a preference list of each switch. The maximum response time of the j_{th} controller is represented as follows:

$$\vartheta(t)_j^{\max} = \frac{1}{\mu_j - \gamma_j \mu_j}. \tag{14}$$

From the above formula, we can see that controllers in the i_{th} switch's preference list $\Gamma(s_i)$ are arranged in ascending order based on their own maximum response time. Switch i prefers the controller whose index is smaller in the preference list. The smaller the index of the controller in the preference list is, the shorter the maximum response time of the controller is. The preference list of the i_{th} switch is shown in

$$\begin{aligned} \Gamma(s_i) &= \{c_{j^*}, \ldots,\}, \quad \forall j \neq j^*, \\ \vartheta(t)_{j^*}^{\max} &< \vartheta(t)_j^{\max}. \end{aligned} \tag{15}$$

From the perspective of the controller, controller j may first choose the switch with smaller request rate and smaller distance between the switch and the controller j. All switches in the j_{th} controller's preference list are arranged in ascending order based on the product of the request rate and the distance between the switch and the controller. Controller j prefers the switch whose index is smaller in the preference list $\Gamma(c_j)$. The smaller the index of the switch in the preference list is, the higher the probability of this switch being selected by the controller j is. The preference list of the j_{th} controller is shown in (16).

$$\begin{aligned} \Gamma(c_j) &= \{s_{i^*}, \ldots,\}, \quad \forall i \neq i^*, \\ \upsilon(t)_{i^*} * d_{i^* j} &< \upsilon(t)_i * d_{ij}. \end{aligned} \tag{16}$$

Of course, in the above case, if the controller j wants to place the i^* switch into the initial mapping, the controller j needs to satisfy the constraint that the controller load can not exceed its maximum threshold after placing the i^* switch into the mapping:

$$\theta(t)_j + \upsilon(t)_{i^*} \leq \gamma_j \mu_j, \quad \forall s_{i^*}. \tag{17}$$

The pseudocode of "bidirectional selection" mechanism is shown in Algorithm 2. Specifically, after each side builds

Input: Initial matching Θ; $\forall j, \mu_j, \gamma_j$
Output: $y(t)_{ij}$, $\forall i, j$
(1) **function** Transfer(Θ, μ_j, γ_j)
(2) **for** All controllers, $\forall j, c_j$ **do**
(3) **for** Each switch $s_i \in \Theta(c_j)$ **do**
(4) **for** controller $c_m \in C \backslash \{c_j\}$ **do**
(5) Find the transfer pair (s_i, c_j, c_m) with minimum TR(s_i, c_j, c_m)
(6) **endfor**
(7) **if** TR$(s_i, c_j, c_m) < 0$ **then**
(8) **Update**: $\Theta \leftarrow$ transfer(s_i, c_j, c_m)
(9) **end if**
(10) **end for**
(11) **end for**
(12) Transform Θ to $y(t)_{ij}$
(13) **end function**

Algorithm 3: Optimal mapping algorithm.

the preference list based on the above definitions, switches start to propose to their most preferred controllers. When the controller receives the proposals, it begins to choose its most preferred switches under the capacity constraint and reject the rest. Repeat this process until there are no more proposals.

4.2.2. Optimal Mapping Algorithm. We can get the initial controller-to-switch mapping Θ by "bidirectional selection" mechanism. However, the initial mapping is not the optimal solution. Therefore, this subsection defines the transfer rules and uses the iterative algorithm (Algorithm 3) to obtain the optimal mapping between the controller and the switch.

Transfer Rule. Assume that switch s corresponds to the controller a in the initial mapping Θ. After satisfying certain transfer rules, switch s corresponds to controller b in the updated mapping. The above process is denoted by transfer(s, a, b). Before the transfer process, s is included in the set of switches mapped by controller a. After the transfer process, s is deleted from the set of switches mapped by controller a and added to the set of switches mapped by controller b. We define the transfer rule based on the average global controller response time. After the transfer process, if the mapping reduces the global controller response time, we call that this process satisfies the transfer rules and then update the controller-to-switch mapping. The above mathematical expression of the transfer process is as follows:

Before transfer(s, a, b)

$$S_a = \Theta(a); \quad S_b = \Theta(b). \tag{18}$$

After transfer(s, a, b)

$$S_{a^*} = \Theta(a^*) = S_a \setminus \{s\}; \quad S_{b^*} = \Theta(b^*) = S_b \cup \{s\}. \tag{19}$$

Transfer Rule

$$\mathrm{TR}(s, a, b) = \vartheta(t)_{a^*} + \vartheta(t)_{b^*} - \left[\vartheta(t)_a + \vartheta(t)_b\right] < 0. \tag{20}$$

According to the above transfer rules, we design the following algorithm to dynamically adjust the controller-to-switch mapping [30] and obtain the transfer pair with minimum transfer rule value TR(s_i, c_j, c_m). After finding the target transfer pair, we update the mapping to achieve the global controller response time optimization.

5. Evaluation of SDNManager

SDNManager and the dynamic controller scheduling strategy are described in detail in Sections 3 and 4. In this section, we will introduce our implementation of SDNManager and evaluate the performance and overhead of our system. Below we will detail the experimental program and analyze the experimental results.

5.1. Implementation. We implement SDNManager and test it under OpenFlow environment. SDNManager is a fast and lightweight denial-of-service detection and mitigation system for SDN. The detailed design is stated in Sections 3 and 4. Figure 4 describes the topology used for the experiments. We use eight physical servers and four pica8 switches in the experiments. Each server is equipped with two Intel(R) Xeon(R) CPU x5690 3.47 GHz and 48 GB of RAM and runs CentOS 6. The 1_{th}~4_{th} server uses virtual machines to run 25 clients (including attackers), respectively. Four Floodlight controllers are implemented on 5_{th}~8_{th} server independently.

Three sets of experiments are performed. The first experiment shows a comparative performance of the forecast engine with DTS model and ARCH model for a sample trace from the network traffic. In the second experiment, we measure the bandwidth received by the legitimate client and the attacker, with and without SDNManager, as a way of validating the prototype. In the meanwhile, we also compare the defense effects of SDNManager, FloodGuard [12], and SGuard [13]. In the last experiment, we measure the CPU

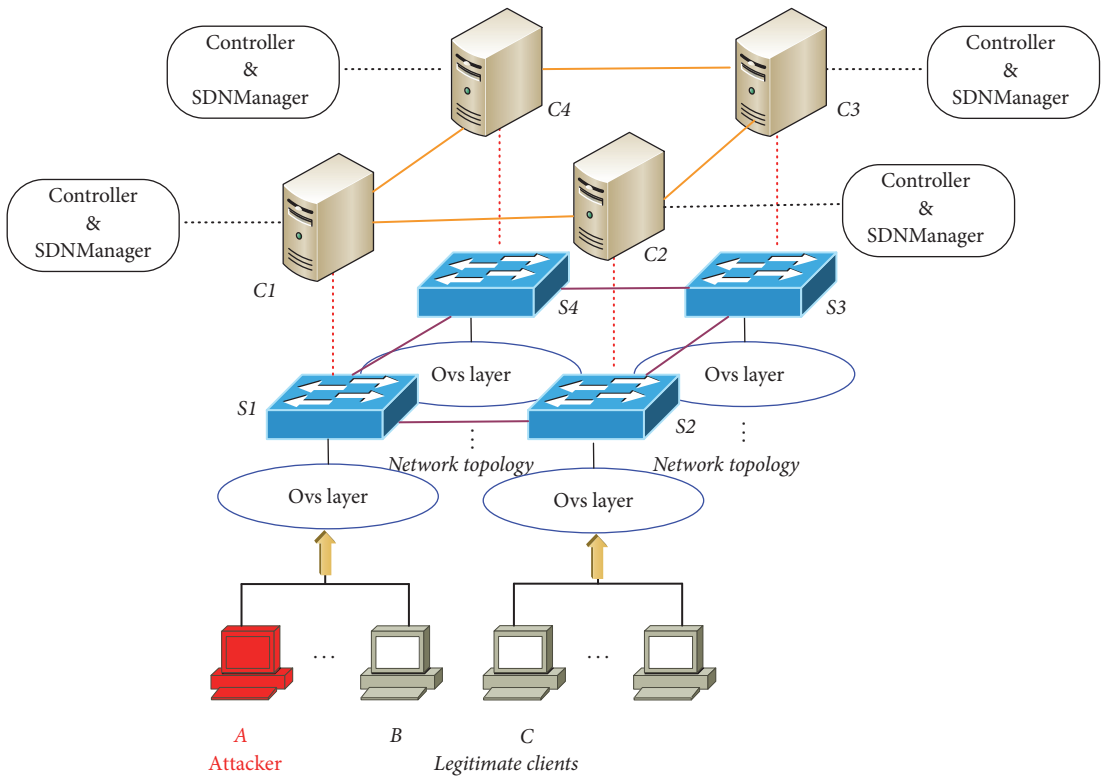


Figure 4: The topology used in the experiments.

Table 1: Keenan test and MAPE for DtS and ARCH.

Trace	Keenan test	MAPE	
	p value	*DTS*	*ARCH*
1 [20]	6.87×10^{-31}	5.23	9.27
2 [21]	2.93×10^{-18}	8.61	11.92

utilization to compare the overhead of SDNManager, SGuard, and FloodGuard.

5.2. Forecast Effect of Forecast Engine. To validate the forecast engine discussed, we conduct statistical analysis by applying it to selected traces from the research [20] and online resource [21]. What is more, in order to enhance persuasiveness, we first conduct it on the traces and testify stationarity and nonlinearity and then compare the performance of the forecast engine with our DTS model and ARCH model. Nonlinearity testing was done with the well-known Keenan test (a low p value is indicative of nonlinearity). We also compute MAPE (Mean Absolute Percentage Error) [31] to achieve a comparison between the two models.

Table 1 summarizes the Keenan test results and MAPE value for each model. From the results, we can see our model satisfies underlying statistical assumptions, which lays the foundation for the correctness of the following experimental results. Figure 5 shows the comparative performance results (sample trace [21], September 5, 2005, 12:28:15–12:29:55).

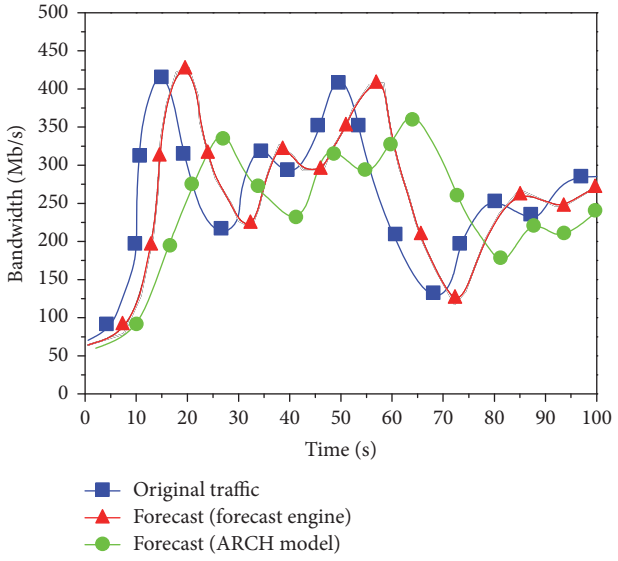


Figure 5: DTS and ARCH forecast of sample trace.

When it comes to forecasting outlier, the DTS model is much more efficient than the ARCH model. We use the following equation to compute forecast error.

$$\text{ForecastError}\,(\%) = \frac{100}{n}\sum_{t=1}^{n}\left|\frac{z_t - \widetilde{z}_t}{z_t}\right|. \tag{21}$$

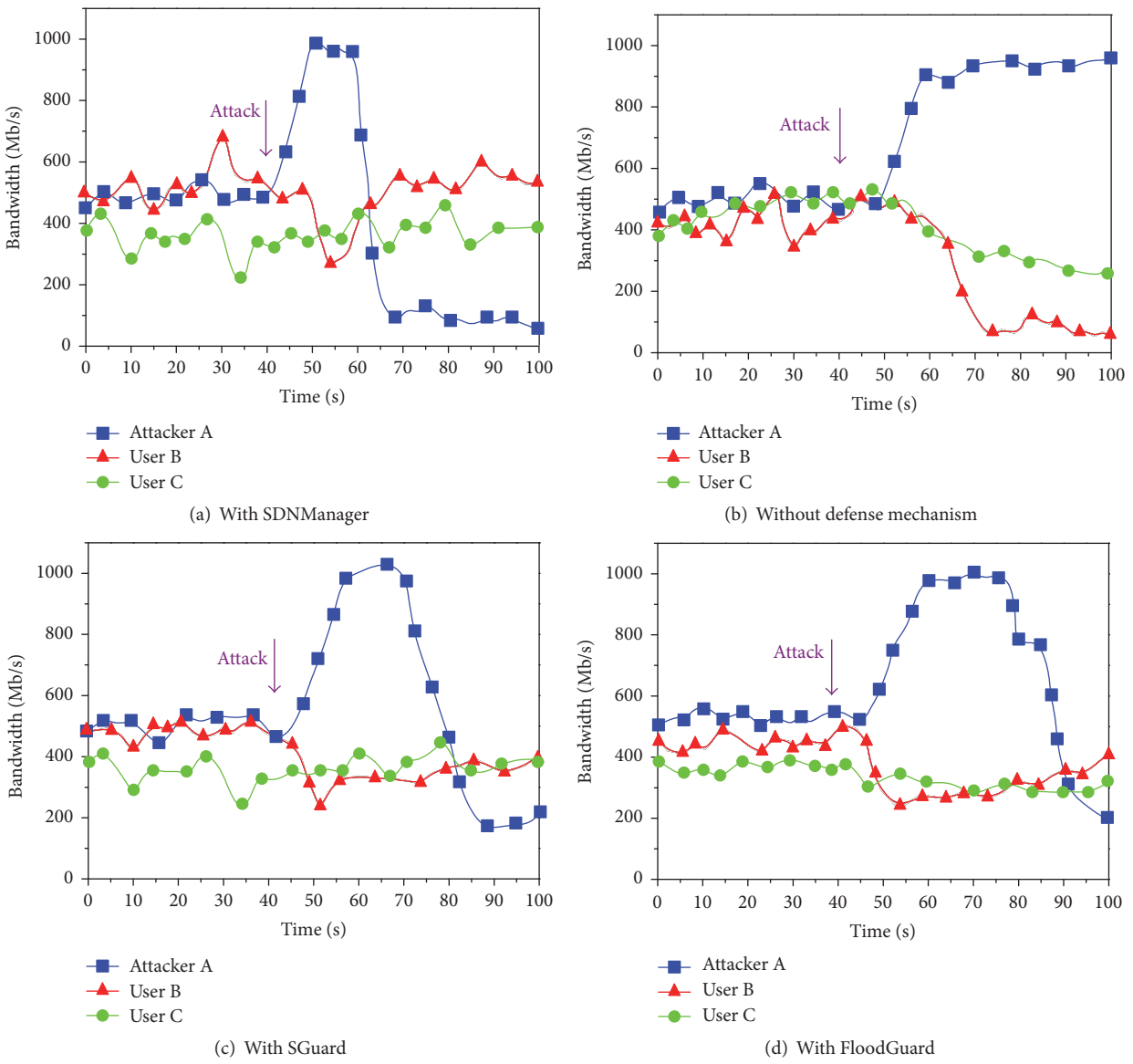


FIGURE 6: The bandwidth changes before and after the attack.

Here, z_t and $\tilde{z}_t$ are the real and estimated values of the time-series. Specifically, the forecast error of DTS can be effectually controlled between 8% and 13%. However, that of ARCH is 20% to 40%. From this perspective, we can conclude that the accuracy of DTS model is higher than that of ARCH model, which is helpful to defend against possible DoS attacks.

5.3. SDNManager Defense Effects. In the second experiment, we measure the bandwidth received by the legitimate client and the attacker, with and without SDNManager, respectively, as a way of validating the prototype. For ease of explanation, we randomly choose three users A, B, and C (as Figure 4 shows) from the networks and assume that A is the attacker, whose attack rate can be up to 1 Gb/s; B and C are normal users. It is also worth noting that attacker A and user B are controlled by the same SDN controller, but user C is controlled by another different controller. Therefore, we can compare the bandwidth received by different legitimate clients so as to increase the reliability of the results. In addition, the randomness introduced in the experiment also increased the reliability of the experimental results. Figures 6(a) and 6(b) show the bandwidth changes of A, B, and C, before and after the saturation attack, with SDNManager and without defense mechanism separately.

In this experiment, with and without SDNManager, all users' bandwidth changes (no matter it is of the attacker or the normal users) are within the normal range when there is no saturation attack (0~40 s). This proves that SDNManager does not harm the bandwidth of traffic forwarding. If we start the saturation attack (at about 40 s) without SDNManager, the bandwidths of both user B and user C go down quickly, whereas the bandwidth received by attacker A continues to increase without any limits. The attacker can easily consume

the computation resource of the controller and overload the infrastructure of SDN networks, and cause the cascading failures of controllers. However, while using SDNManager the saturation attack also starts at about 40 s, and the bandwidth of user B firstly decreases a little and then returns to normal; the bandwidth of user C almost remains unchanged. This is because flows with bandwidth consumption higher than the predicted bandwidth consumption will be penalized by the application. The penalization is proportional to the difference between current and predicted usage. In addition, the forecast engine of SDNManager employs a novel dynamic time-series (DTS) model which greatly improves bandwidth prediction accuracy. In this case, SDNManager can protect the SDN significantly and avoid the cascading failures of controllers effectively.

In order to compare the defense effect of SDNManager with the other defense mechanism, we implement SGuard and FloodGuard on the same physical environment. In the meanwhile, we also measure the bandwidth received by the legitimate client and the attacker. Figures 6(c) and 6(d) show the bandwidth changes of A, B, and C, before and after the saturation attack, with SGuard and FloodGuard separately.

With SGuard and FloodGuard, all users' bandwidth slightly decreases (no matter it is of the attacker or the normal users) when there is no saturation attack (0~40 s). In this process, SGuard receives collected flows, extracts features that are important to the DoS flooding attack detection, and gathers them in 6-tuples which would be passed to the classifier module. Then, the classifier module analyzes whether a given 6-tuple corresponds to a DoS flooding attack or legitimate traffic. FloodGuard also needs to analyze data plane messages and update its packet-processing policies in this process. Therefore, SGuard and FloodGuard need to make a comprehensive judgment according to multifactors and then take the appropriate protective measures. This process involves a lot of complex calculations, so the evaluation results are not surprising. This also proves that both SGuard and FloodGuard have a slightly negative impact on the bandwidth of traffic forwarding. If we start the saturation attack (at about 40 s), the bandwidths of both user B and user C go down slowly, whereas the bandwidth received by attacker A slightly increases. The attacker can hardly consume the computation resource of the controller and overload the infrastructure of SDN networks, as well as cause the cascading failures of controllers. This proves that both SGuard and FloodGuard have certain defense effects. More specifically, when the saturation attack is detected, FloodGuard's proactive flow rule analyzer module dynamically tracks the runtime value of the state sensitive variables from the running applications, converts generated path conditions to the proactive flow rules dynamically, and installs these flow rules into the OpenFlow switches. SGuard can also classify traffic as either normal or an attack by using the features extracted from the data set and then take some measures to block attackers. However, the defense effects of SGuard and FloodGuard are worse than the defense effect of SDNManager. For example, when we use SGuard and FloodGuard, the bandwidth received by the legitimate client is lower than that in SDNManager, while the bandwidth received by the attacker is higher

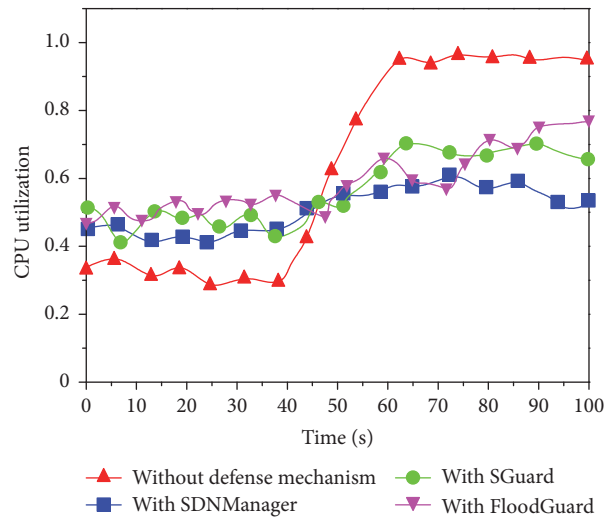


Figure 7: CPU utilization.

than that in SDNManger. In addition, the attack detection time of SDNManager is also less than that of SGuard or FloodGuard (SDNManager: about 15 s, SGuard: about 28 s, and FloodGuard: about 37 s). SDNManager uses a lightweight DTS model to predict the bandwidth consumption. The penalization is also based on the difference between current and predicted usage. Thus, SDNManager can quickly detect the DoS attacks. Prevention and early detection of DoS attack are very important. SDNManager can minimize the delay of detecting DoS attack after its occurrence. From this perspective, we can conclude that SDNManager has better defense effects than SGuard and FloodGuard.

5.4. Overhead Analysis. In this section, we show our evaluation about the overhead of SDNManager. With SDNManager and without SDNManager, we keep monitoring the resource consumption of controller 1 (we choose the CPU utilization of each controller as the indicator of how many resources it consumes). Figure 7 shows the evaluation results. We can observe that when there is no attack (before the 40 s), the CPU utilization of controller with SDNManager is a little higher than that of the controller without any defense mechanism. Not surprisingly, this is owing to the design of SDNManager. SDNManager is responsible for performing a bandwidth prediction algorithm so that it will have a little impact on controller efficiency, but this impact is still within our expected tolerance. When we start the saturation attack at about 40 s, the CPU utilization of controller with SDNManager almost remains unchanged, while that of the controller without defense mechanism increases quickly and reaches a peak at about 50 s. Thus, the overhead of SDNManager is acceptable for our system. We also compare and analyze the overhead of SDNManager, SGuard, and FloodGuard. We continue to use CPU utilization to represent the overhead of the system. It is obvious to see that the overall utilization of SDNManager is relatively low, which shows that SDNManager is highly scalable and able to provide security services for more network devices. In contrast, the overhead of SGuard and

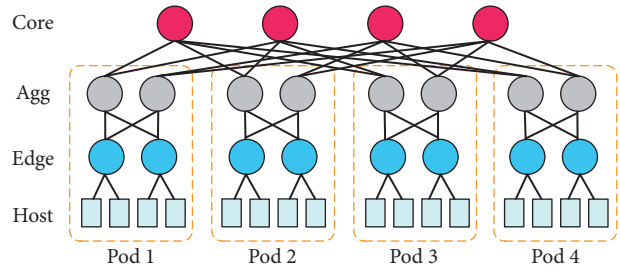


Figure 8: 4-pod fat-tree (example topology).

FloodGuard is higher. SGuard and FloodGuard need to make a comprehensive judgment according to multifactors and then take the appropriate protective measures. This process involves a lot of complex calculations, so the evaluation results are not surprising.

6. Evaluation of DCS Model

Through the previous analysis, we can see that the SDNManager proposed in this paper has obvious advantages compared with other defense mechanisms, such as SGuard and FloodGuard. However, the experimental environment in Section 5 is relatively static, which cannot show the advantage of the controller dynamic scheduling strategy applied in the cloud data center to the global network performance [32]. Therefore, in order to test the deployment effect of the controller dynamic scheduling strategy in the actual SDN environment, this section further deploys the strategy in the experimental cloud data center to test its defense effect.

The data center topology chosen in this section is a 24-pod fat-tree structure (Figure 8) with a total of 720 switches and 3456 host users. There are 30 Floodlight controllers deployed in this data center. The decay factor of each controller γ_i is set to (0.92~0.96) randomly. All Floodlight controllers are implemented on different physical servers independently. Each Server is equipped with two Intel(R) Xeon(R) CPU x5690 3.47 GHz and 48 GB of RAM and runs CentOS 6. Out-of-Band control uses a separate network to connect all the switches with the controller. We implement SDNManager, SGuard, and FloodGuard on each controller. The attackers in the cloud data center are randomly selected which are five percent of the total number of users. The other experimental parameters are set as shown in Section 5. When attack rate is 100 Mb/s, 200 Mb/s, 400 Mb/s, and 800 Mb/s, respectively, we test and analyze the average controller response time of SDNManager, SGuard, and FloodGuard without applying controller dynamic scheduling strategy. In order to facilitate the comparative analysis, we repeat the experiment with applying controller scheduling strategy. The experimental results are shown in Figures 9 and 10. Figures 9(a)–9(d) show the global controller response time at different attack rates when the controller dynamic scheduling strategy is not applied, and Figures 10(a)–10(d) show the global controller response time at different attack rates when the controller dynamic scheduling strategy is applied.

As can be seen from Figures 9(a)–9(d) in the absence of controller dynamic scheduling strategy, the average controller response time of SDNManager is significantly lower than that of SGuard and FloodGuard before launching attacks ($t < 10$ s). On the one hand, the reason is that the overhead of SDNManager is less than that of SGuard and FloodGuard. On the other hand, SGuard and FloodGuard are relatively less scalable (Scalability is the measure of how a system responds when additional hardware is added). It is worth noting that the size of the second experimental topology is significantly larger than that of the first experimental topology. When we expand the topology, the large topology brings a heavy load to SGuard's abnormal traffic detection module and FloodGuard's proactive flow rule analyzer module. More specifically, SGuard has to receive much more collected flows, extract features that are important to DoS flooding attack detection, and gather them in 6 tuples which are passed to the classifier module to analyze whether a given 6-tuple corresponds to a DoS flooding attack or legitimate traffic. FloodGuard's proactive flow rule analyzer module must combine symbolic execution and dynamic application tracking to derive proactive flow rules at runtime. This process involves a lot of complex calculations and greatly increase the response time. In contrast, SDNManager only needs to predict the bandwidth consumption and enforce the penalization strategy based on the difference between current and predicted usage, which greatly reduce the consumption of resources. After $t = 10$ s, when attackers begin to launch attacks, the average response time of SGuard and FloodGuard gradually increases. With the increase of attack rate, the average response time also increases (Δt[lower Mb·s^{-1}] < Δt[higher Mb·s^{-1}]). When attack rate is 100 Mb/s, 200 Mb/s, 400 Mb/s, and 800 Mb/s, respectively, the corresponding peak response time of SGuard and FloodGuard is {(0.7, 0.85); (0.9, 1.2); (1.16, 1.62); (2, 2.6)}. From the above results, although the response time is affected, it is still within a reasonable range. Compared to the above two defense mechanisms, SDNManager has some advantages regarding overhead. For example, the average response time of SDNManager is basically within (0.2, 0.6) even at different attack rates and it also fluctuates smoothly in some ranges. It proves that SDNManager is a lightweight and fast denial-of-service detection and mitigation system for SDN again.

As can be seen from Figures 10(a)–10(d), when we use the controller dynamic scheduling strategy, the average controller response time of SDNManager, SGuard, and FloodGuard significantly decreases before launching attacks ($t < 10$ s). This is because the proposed controller dynamic scheduling strategy can effectively balance the data center controller load and optimize the global response time. After $t = 10$ s, when attackers begin to launch attacks, the average response time of SDNManager, SGuard, and FloodGuard gradually increases. With the increase of attack rate, the average response time also increases (Δt[lower Mb·s^{-1}] < Δt[higher Mb·s^{-1}]). When attack rate is 100 Mb/s, 200 Mb/s, 400 Mb/s, and 800 Mb/s, respectively, the corresponding peak response time of SDNManager, SGuard, and FloodGuard is {(0.15, 0.46, 0.65); (0.38, 0.52, 0.85); (0.41, 0.79, 1.17); (0.48, 1.13, 1.72)}. It can be seen from the above results that, in

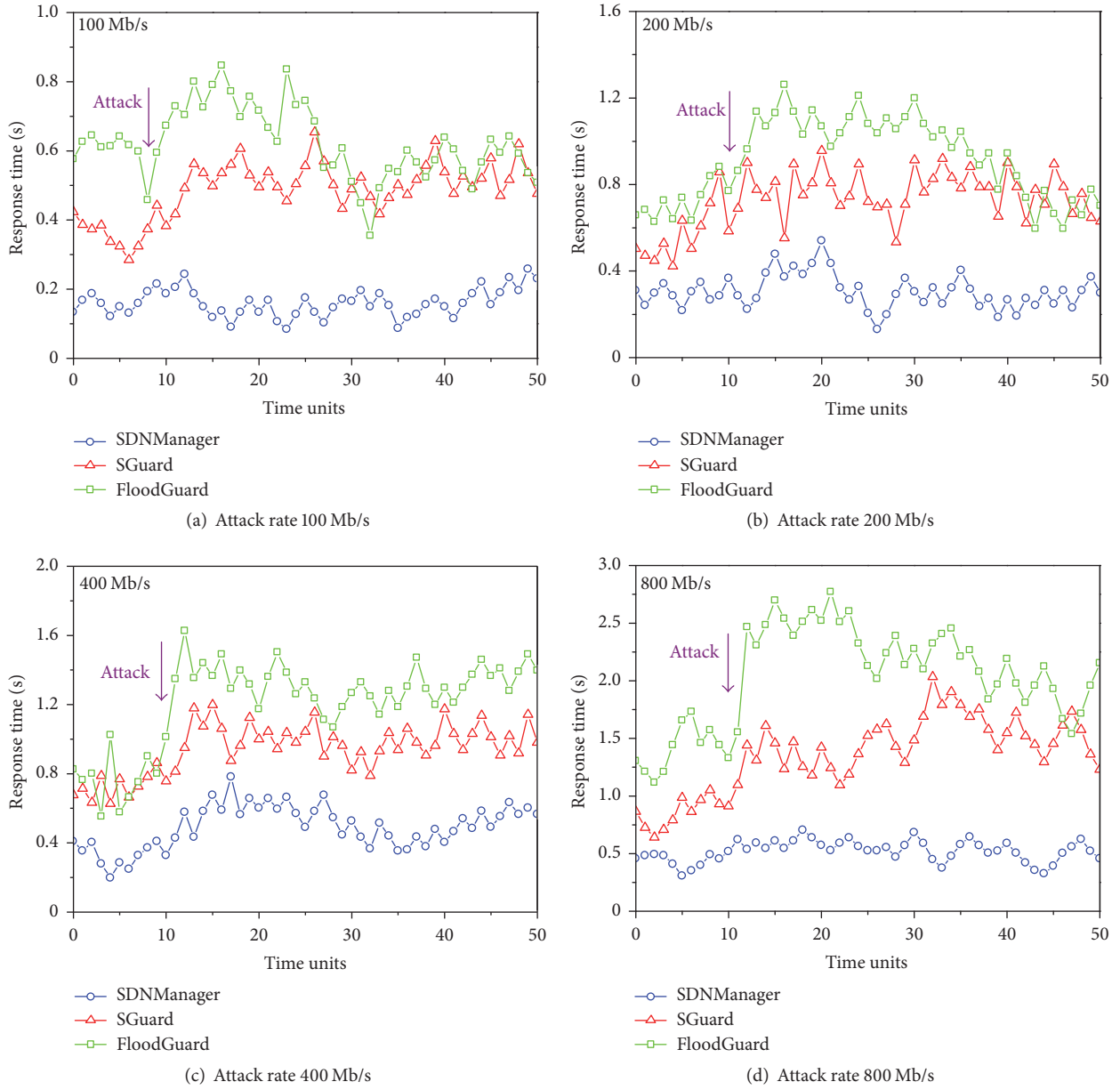


Figure 9: Response time comparison using dynamic controller scheduling strategy.

the latter case, the peak response time is significantly reduced. Less statistical fluctuation in response time also produces a more consistent end-user experience. Overhead refers to the processing time required by system software, which includes the operating system and any utility that supports application programs. Response time is the total amount of time it takes to respond to a request for service. Thus, response time can indicate the overhead of the system. For a given request the service time varies little as the workload increases. As can be seen from Figures 9 and 10, the response time with dynamic controller scheduling strategy is lower than that without the dynamic controller scheduling strategy. On the one hand, it proves that the dynamic controller scheduling strategy can significantly optimize the average controller response time. On the other hand, it proves that the overhead of strategy is in a reasonable range.

In summary, in this experimental cloud data center scenario, the controller dynamic scheduling strategy proposed in this paper significantly optimizes the average controller response time, which achieves the expected target.

7. Future Work

In the future, we plan to do the following two works.

First, in order to expand the scale and scope of SDNManager, we plan to implement it on Ryu or OpenDaylight

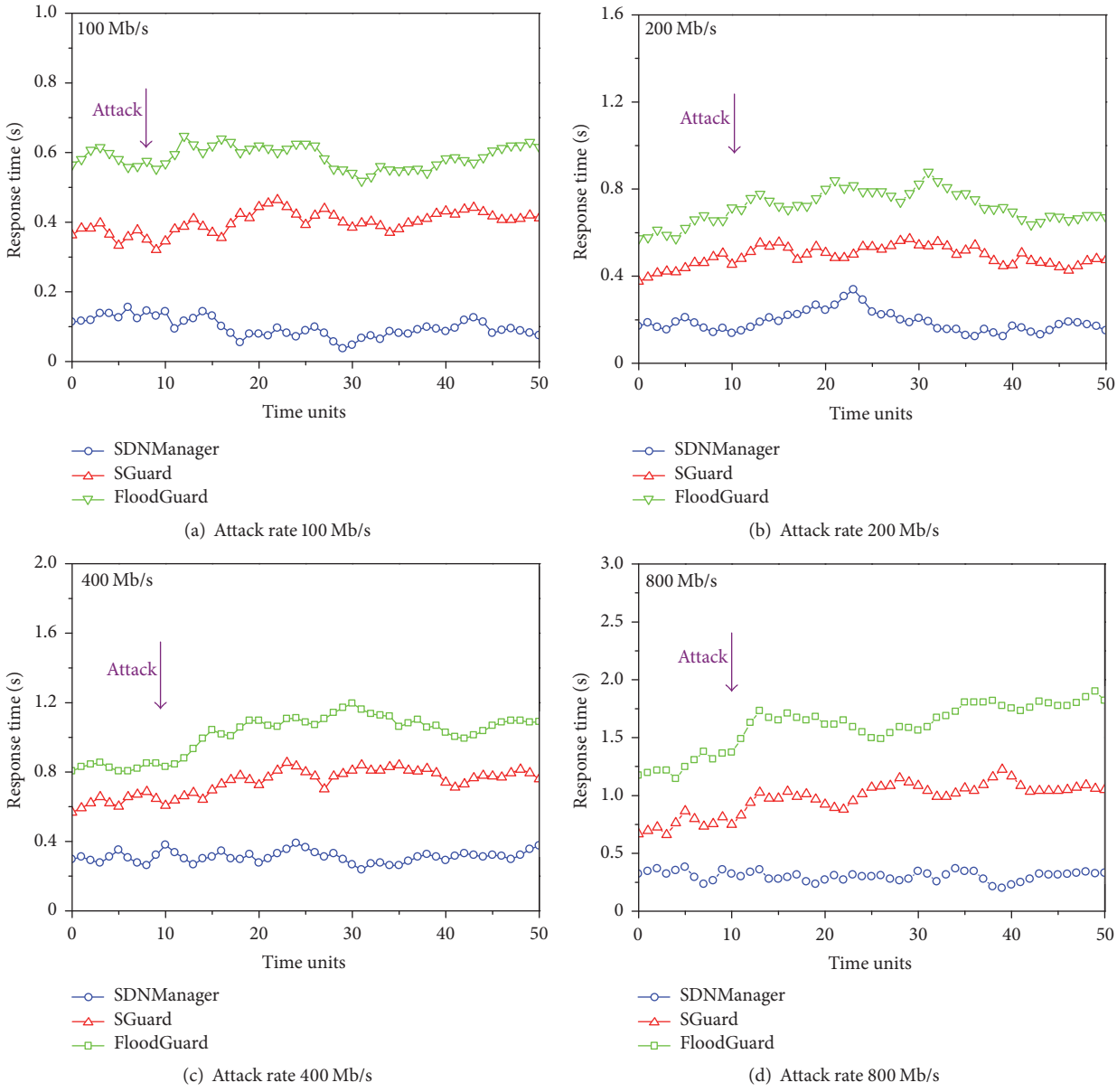


Figure 10: Response time comparison using dynamic controller scheduling strategy.

in the future. There is no doubt that the current popular SDN controllers are Ryu and OpenDaylight. Ryu is commonly referred to as component-based, open source software defined by a networking framework, which is implemented entirely in Python. It can provide software components with well-defined APIs that are exposed to allow developers to create new network management and control applications. It also supports multiple southbound protocols for managing devices. What is more, OpenDaylight is also a highly available, modular, extensible, scalable, and multiprotocol controller infrastructure built for SDN deployments on modern heterogeneous multivendor networks, which provides a model-driven service abstraction platform that allows users to write apps that easily work across a wide variety of hardware and southbound protocols. Therefore, Ryu and OpenDaylight are two of those SDN controllers that we as developers should seriously consider. Above all, if we conduct the science experiment on Ryu or OpenDaylight, the experimental results will be better.

Second, we plan to combine SDNManager and the existing Intrusion Detection System (IDS) to further improve defense efficiency. In this paper, we view SDN DoS attack as a resource management problem. It is a cyber-attack where the perpetrator seeks to make the controller resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the controller. It is typically accomplished by flooding the targeted controller with superfluous requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled. Similarly, burst traffic may also cause network congestion

and cause the packet drop to take place, thus reducing the overall throughput. Therefore, it is difficult to distinguish normal burst traffic and DoS attack traffic. A more detailed classification requires an Intrusion Detection System (IDS), which would be considered in the future.

8. Conclusions

In this paper, we propose SDNManager to prevent SDN DoS attacks and the cascading failures of controllers. SDNManager follows a control loop of reading flow statistics, forecasting flow bandwidth changes based on the statistics and updating the network accordingly. Flows with bandwidth consumption higher than a predicted usage are penalized by the application. The penalization is proportional to the difference between current and predicted usage. Thus, attackers are served with a lower priority than the normal users. The evaluation results demonstrate the effectiveness of SDNManager and show that our system only adds minor overhead.

Notations of SDNManager

C_i:	i_{th} controller
flow_{ij}:	i_{th} flow is corresponding to i_{th} controller
BWflow_{ij}:	Current bandwidth utilization of flow_{ij}
$\text{BW}_{\text{flow}_{ij}.\text{fcst}}$:	Predicted bandwidth utilization of flow_{ij}
$\text{BW}_{\text{flow}_{ij}.\text{target}}$:	Target allocation bandwidth of flow_{ij}
OS_{c_i}:	The observed state variables
PS_{c_i}:	The proposed state variables
TS_{c_i}:	The target state variables
f_t:	A time-varying parameter
θ_t:	Parameter of the conditional distribution of bandwidth
μ:	Decay factor of control-to-data path
s_t:	The conditional score
ξ:	Slack variable.

Conflicts of Interest

The authors declare there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (no. 060802, 61521003), the National key Research and Development Program of China (nos. 2016YFB0800100, 2016YFB0800101), the National Natural Science Foundation for Young Scientists (no. 61602509), the Science and Technology Project of Henan Province (172102210615), and the Emerging Direction Fostering Fund of Information Engineering University (2016610708).

References

[1] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Revie*, vol. 38, no. 2, pp. 69–74, 2008.

[2] S. Shin, V. Yegneswaran, P. Porras et al., "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 413–424, Berlin, Germany, 2013.

[3] T. Wang, F. Liu, J. Guo et al., "Dynamic SDN controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, San Francisco, Calif, USA, 2016.

[4] K. ElDefrawy and T. Kaczmarek, "Byzantine fault tolerant software-defined networking (SDN) controllers," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 208–213, Atlanta, Ga, USA, 2016.

[5] G. Yao, J. Bi, and L. Guo, "On the cascading failures of multi-controllers in software defined networks," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Goettingen, Germany, 2013.

[6] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 623–654, 2016.

[7] I. Ahmad, S. Namal, M. Ylianttila et al., "Security in software defined networks: a survey," *IEEE Communications Surveys& Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[8] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*, pp. 151-152, Hong Kong, China, 2013.

[9] Q. Yan and F. R. Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 52–59, 2015.

[10] D. Kotani and Y. Okabe, "A packet-in message filtering mechanism for protection of control plane in OpenFlow networks," in *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2014*, pp. 29–40, Los Angeles, Calif, USA, October 2014.

[11] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proceedings of the 2015 International Conference on Computing, Networking and Communications, ICNC 2015*, pp. 77–81, Garden Grove, Calif, USA, February 2015.

[12] H. Wang, L. Xu, and G. Gu, "FloodGuard: a DoS attack prevention extension in software-defined networks," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 239–250, Rio de Janeiro, Brazil, 2015.

[13] T. Wang and H. Chen, "SGuard: a lightweight SDN safe-guard architecture for DoS attacks," *China Communications*, vol. 14, no. 6, pp. 113–125, 2017.

[14] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *IEEE Local Computer Network Conference*, pp. 408–415, Denver, Colo, USA, 2010.

[15] Q. Yan, Q. Gong, and F.-A. Deng, "Detection of DDoS attacks against wireless SDN controllers based on the fuzzy synthetic evaluation decision-making model," *Ad Hoc & Sensor Wireless Networks*, vol. 33, no. 1, pp. 275–299, 2016.

[16] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic SDN controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017.

[17] W. Wei, X. Wei, T. Chen et al., "Dynamic correlative VM placement for quality-assured cloud service," in *2013 IEEE*

International Conference on Communications (ICC), pp. 2573–2577, IEEE, Budapest, Hungary, 2013.

[18] A. Amin, A. Colman, and L. Grunske, "An approach to forecasting QoS attributes of web services based on ARIMA and GARCH models," in *Proceedings of the 2012 IEEE 19th International Conference on Web Services, ICWS 2012*, pp. 74–81, Honolulu, Hawaii, USA, 2012.

[19] B. Krithikaivasan, Y. Zeng, K. Deka et al., "ARCH-based traffic forecasting and dynamic bandwidth provisioning for periodically measured nonstationary traffic," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 683–696, 2007.

[20] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*, pp. 267–280, Melbourne, Australia, 2010.

[21] "University of Napoli, Network Monitoring and Measurements." http://traffic.comics.unina.it/Traces/ttraces.php.

[22] F. X. Diebold and M. Nerlove, "The dynamics of exchange rate volatility: A multivariate latent factor ARCH model," *Journal of Applied Econometrics*, vol. 4, no. 1, pp. 1–21, 1989.

[23] J. A. Cornell, "Fitting a slack-variable model to mixture data: some questions raise," *Journal of Quality Technology*, vol. 32, no. 2, pp. 133–147, 2000.

[24] M. Kuerban, Y. Tian, Q. Yang et al., "DOS attack mitigation strategy on SDN controller," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Long Beach, Calif, USA, 2016.

[25] A. Roy, H. Zengy, J. Baggay et al., "Inside the social network's (datacenter) network," in *The 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 123–137, London, United Kingdom, 2015.

[26] M. Yu, J. Rexford, M. J. Freedman et al., "Scalable flow-based networking with DIFANE," in *Proceedings of the 7th International Conference on Autonomic Computing, SIGCOMM 2010*, pp. 351–362, New Delhi, India, 2010.

[27] T. Koponen, M. Casado, N. Gude et al., "Onix: a distributed control platform for large-scale production networks," in *Usenix Conference on Operating Systems Design and Implementation. USENIX Association*, pp. 351–364, 2010.

[28] A. Tootoonchian, S. Gorbunov, Y. Ganjali et al., "On controller performance in software-defined networks," in *Usenix Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, pp. 10-10, 2012.

[29] F. Liu, J. Guo, X. Huang et al., "EBA: efficient bandwidth guarantee under traffic variability in datacenters," *IEEE/ACM Transactions on Networking*, vol. 25, no. 1, pp. 506–519, 2017.

[30] J. Guo, F. Liu, D. Zeng et al., "A cooperative game based allocation for sharing data center networks," in *2013 Proceedings IEEE INFOCOM*, pp. 2139–2147, Turin, Italy, 2013.

[31] D. M. Keenan, "A tukey nonadditivity-type test for time series nonlinearity," *Biometrika*, vol. 72, no. 1, pp. 39–44, 1985.

[32] Z. Cai, Z. Wang, K. Zheng et al., "A Distributed TCAM coprocessor architecture for integrated longest prefix matching, policy filtering, and content filtering," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 417–427, 2013.

Hindawi
Security and Communication Networks
Volume 2018, Article ID 4760632, 15 pages
https://doi.org/10.1155/2018/4760632

Research Article

Exploiting the Vulnerability of Flow Table Overflow in Software-Defined Network: Attack Model, Evaluation, and Defense

Yadong Zhou,[1] **Kaiyue Chen**,[1] **Junjie Zhang**,[2] **Junyuan Leng**,[1] **and Yazhe Tang**[1]

[1]*MOE Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China*
[2]*Department of Computer Science and Engineering, Wright State University, Fairborn, OH, USA*

Correspondence should be addressed to Yadong Zhou; yadongzhou@gmail.com

Received 28 September 2017; Accepted 6 December 2017; Published 9 January 2018

Academic Editor: Zhiping Cai

As the most competitive solution for next-generation network, SDN and its dominant implementation OpenFlow are attracting more and more interests. But besides convenience and flexibility, SDN/OpenFlow also introduces new kinds of limitations and security issues. Of these limitations, the most obvious and maybe the most neglected one is the flow table capacity of SDN/OpenFlow switches. In this paper, we proposed a novel inference attack targeting at SDN/OpenFlow network, which is motivated by the limited flow table capacities of SDN/OpenFlow switches and the following measurable network performance decrease resulting from frequent interactions between data and control plane when the flow table is full. To the best of our knowledge, this is the first proposed inference attack model of this kind for SDN/OpenFlow. We implemented an inference attack framework according to our model and examined its efficiency and accuracy. The evaluation results demonstrate that our framework can infer the network parameters (flow table capacity and usage) with an accuracy of 80% or higher. We also proposed two possible defense strategies for the discovered vulnerability, including routing aggregation algorithm and multilevel flow table architecture. These findings give us a deeper understanding of SDN/OpenFlow limitations and serve as guidelines to future improvements of SDN/OpenFlow.

1. Introduction

By decoupling the control plane from the data plane, Software-Defined Network (SDN) makes programmability a built-in feature for networks, thereby introducing automaticity and flexibility to the networking management. SDN has therefore been foreseen as the key technology that enables the next generation of networking paradigm. Despite its promise, one of the most significant barriers towards SDN's wide practical deployment resides in overwhelming security concerns [1]. Therefore, proactively detecting, quantifying, and mitigating its security vulnerabilities become of fundamental importance.

In spite of its novelty, SDN indeed reuses various design and implementation elements ranging from architectures and protocols to systems from traditional network. It is not surprising that SDN inheres the vulnerabilities intrinsic to these elements. For example, similar to any networked service, secure channels between controllers and switches might be disrupted by DDoS attacks; like firewall rules, the flow entries may also conflict with each other, leaking unwanted traffic; malicious arp spoofing generated by attackers may poison the controller MAC table, disturbing the normal topology information gathering and packet forwarding; untrusted applications may instrument SDN controller to perform malicious behaviors without proper access control, which is one of the design objectives for modern operating systems. In response, existing research in the context of SDN security mainly focuses on detecting and mitigating these vulnerabilities. For example, [2] evaluates man-in-the-middle attacks that target at SDN/OpenFlow secure channels; FortNOX [3] brings security enforcement module into NOX [4] and enables real-time flow entry conflict check; VeriFlow [5] detects network-wide invariant violations by acting as a transparent layer between control plane and data plane.

In this paper, we introduce a novel SDN vulnerability. The novelty of this vulnerability stems from the feedback-loop nature of SDN, a fundamental difference compared

with traditional networks. Particularly, this vulnerability can be extremely severe in SDN-based networks where network traffic from different sources shares the same SDN switch's flow table, for example, different tenants in a SDN-based cloud computing network.

Specifically, most commercial SDN/OpenFlow switches have limited flow table capacities, ranging from hundreds to thousands [6]. Such capacity is usually insufficient to handle millions of flows that are typical for enterprise and data center networks [7]. Nevertheless, the flow table capacity was just considered as a potential bottleneck of resource consuming attacks in the past, motivating researches on flow caching systems like [8–10]. But according to our analysis, the flow table capacity can lead to inference attack and privacy leakage under certain circumstances.

As a consequence of flow table overflow, the SDN controller needs to dynamically maintain the flow table by inserting and deleting flow entries. The maintaining process typically includes packet information transferring, routing rule calculation, and flow entry deployment, which leads to measurable network performance decrease.

Particularly, once the flow table is full, extra interactions between controller and switch are needed to remove certain existing flow entries to make room for newly generated flow entries, resulting in further network performance decrease. An attacker can therefore leverage the perceived performance change to deduce the internal state of the SDN. To be more specific, we consider the scenario that an attacker resides in a network that is managed by a SDN. The attacker can then actively generate network traffic, triggering the interactions between the controller and switch with respect to flow entry insertion and deletion. The attacker can then measure the change of the network performance to estimate the internal state of the SDN including the flow table capacity and flow table usage. We have designed innovative algorithms to exploit this vulnerability and quantify their effectiveness on exploiting this vulnerability based on extensive evaluation.

Additionally, to mitigate this vulnerability, we have proposed two possible defense strategies. The first strategy is a new routing aggregation algorithm to compress the flow entries so they will consume less flow table space. The second strategy is building a multilevel flow table architecture. Multilevel flow table architecture can implement flow tables with larger capacities without introducing additional power assumption or charges.

To summarize, in this paper we made the following contributions:

(i) We have identified a novel vulnerability introduced by the limited flow table capacities of SDN/OpenFlow switches and formalized that threat.

(ii) We have designed effective algorithms that can successfully exploit this vulnerability to accurately infer the internal states of the SDN network including flow table capacity and flow table usage.

(iii) We have performed extensive evaluation to quantify the effectiveness of proposed algorithms. The experimental results have demonstrated that the discovered vulnerability indeed leads to significant security concerns: our algorithm can infer the network parameters with an accuracy of 80% or higher across various network settings.

(iv) We have proposed two possible defense strategies for the discovered vulnerability, including routing aggregation algorithm to compress the flow entries, and multilevel flow table architecture to implement flow tables with larger capacities.

The rest of this paper is organized as follows. Section 2 gives an overall statement of the inference attack problem. Section 3 gives detailed inference algorithms targeting at FIFO and LRU replacement algorithms, respectively. Section 4 gives a detailed evaluation of the simulation results. Section 5 proposes two possible defense methods against this kind of inference attack. Section 6 is a brief discussion about our findings and future research. Section 7 describes some related works in this area. Finally, Section 8 concludes this paper.

2. Problem Statement

The vulnerability of flow table overflow in SDN potentially exists in SDN-based cloud computing network and other important SDN-based networking systems [11, 12].

After analyzing current structure and implementation of SDN/OpenFlow, its decoupled nature gives us inspiration: the interactions between control plane and data plane will lead to network performance decrease, which can be measured through performance parameters like round trip time (RTT). If a flow matches one flow entry, the flow will be forwarded directly according to the matched entry. This process is fast and will cost little time. When the flow table is full, some flow entry will be removed, then the controller has to calculate the rule and send a new flow entry to the switch, and this process is more complex and has more interactions between controller and switch than the previous case, which will cost more time.

Figure 1 gives an overall flowchart of packet processing in an OpenFlow switch. The three rectangular regions surrounded by dotted line stand for three possible packet processing branches, respectively. When the switch encounters an incoming packet, it will parse it and send the parsed packet into the subsequent processing pipeline.

Then as the first step of the pipeline, the switch will look up its flow table to search flow entries matching the packet. When there is a match, the switch will directly forward the packet according to actions associated with the corresponding flow entry. This branch is illustrated in the innermost rectangle of Figure 1.

When there is no corresponding flow entry in the flow table, extra steps will be introduced into the procedure. Additional interactions between the switch and the controller will happen to acquire corresponding routing rules, including packet information transferring, routing rule calculation, and flow entry deployment. The middle rectangle of Figure 1 illustrates this process.

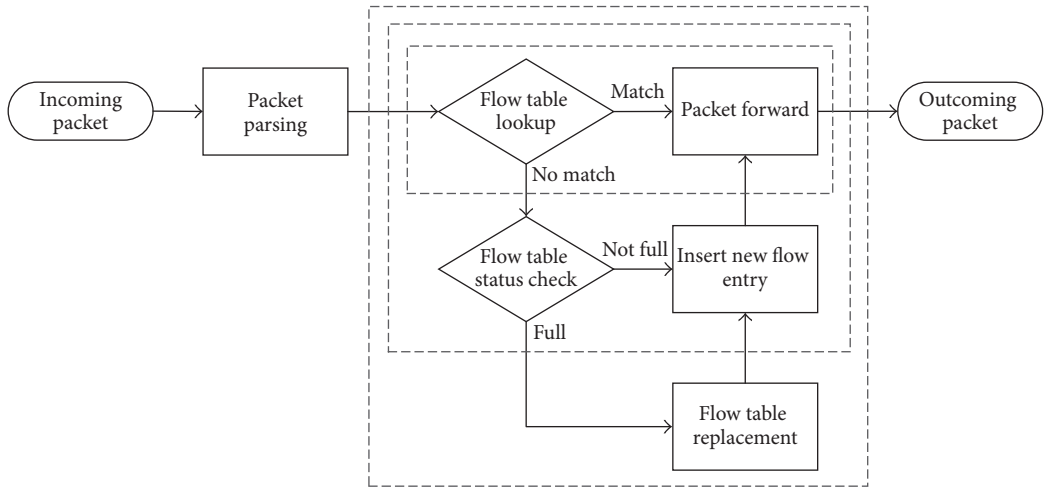


Figure 1: OpenFlow packet processing flowchart.

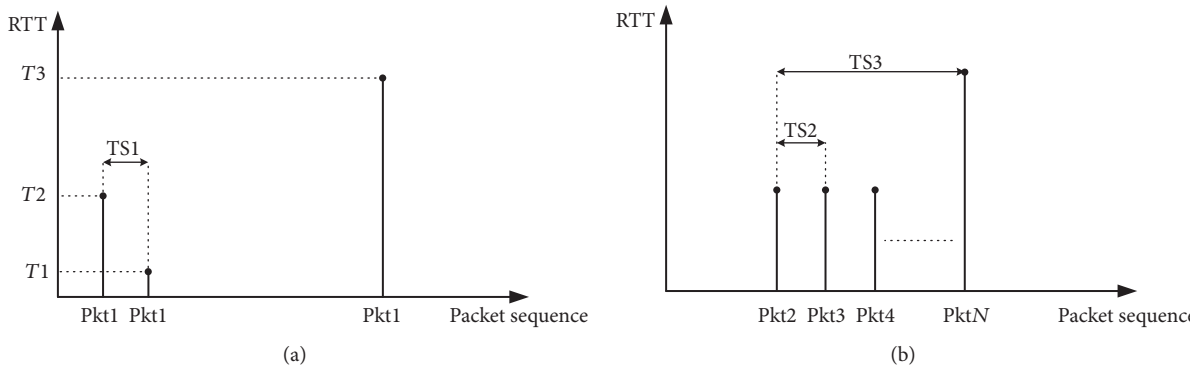


Figure 2: RTT measurement of different flow table state.

Before the switch inserts the newly generated flow entry, it has to check the flow table status to make sure that there is enough space in the flow table. When the flow table is full, the controller has to perform flow table replacement operations to make room for the upcoming flow entry. These operations include deciding which old flow entry to delete according to certain flow table replacement algorithm and flow entry deletion. The outermost rectangle in Figure 1 stands for this branch.

That is exactly where the vulnerability lies. In traditional networks, the switches and routers are autonomous, which means they can maintain their routing tables locally without interacting with an external device. But due to the decoupled nature of SDN/OpenFlow, maintaining switch flow tables needs frequent interactions between switches and controllers, making it possible for an attacker to leverage the perceived performance change to deduce the internal state of the SDN network.

As shown in Figure 1, the rectangular regions surrounded by dotted line correspond to different possible packet processing branches. The larger a rectangle is, the longer the processing time of that branch will be because of the extra steps that rectangle contains. When there is a match in the flow table, the processing time will be the shortest; when there is no match in the flow table and the flow table is not full, the processing time will be longer because of addition routing calculation and flow entry deployment; when there is no match in the flow table and the flow table is full, the processing time will be the longest because a flow table replacement operation has to be performed. So as a network parameter directly influenced by the processing time, the RTT of a packet can serve as an indicator of flow table state and flow entry state.

The process of deciding RTT thresholds for flow table state detection is shown in Figure 2.

Figures 2(a) and 2(b) represent two cooperating threads, the x-axis represents the packet sequence, and the y-axis represents the recorded RTT of every packet. Firstly, in the upper thread, we generate a packet with a specific {src_ip, dst_ip, src_mac, dst_mac} combination, calling it Pkt_1. Send Pkt_1 to the target OpenFlow switch and record the corresponding RTT as T_2. Currently there is no corresponding flow entry in the OpenFlow switch because Pkt_1 is a new packet. After a time span TS_1, send Pkt_1 to the target OpenFlow switch again and record the corresponding RTT as T_1. If TS_1 is chosen properly, the newly installed flow

entry matching Pkt_1 should still exist in the OpenFlow switch. Next, in the lower thread, we continuously generate packets $\text{Pkt}_2, \text{Pkt}_3, \ldots, \text{Pkt}_N$, each with a different combination of {src_ip, dst_ip, src_mac, dst_mac} and send these packets to the target OpenFlow switch with the time span of TS_2. Because there are no flow entries matching their packets in the OpenFlow switch, the recorded RTTs will be approximately the same as T_2. Keep generating and sending packets until we observe a sudden increase of the RTT, which indicates that the flow table is full. Then in the upper thread we send Pkt_1 again immediately and record the RTT as T_3. To achieve higher precision, we can repeat the process and use average values of T_1, T_2, and T_3 as final results.

From the process above we can see that T_1, T_2, and T_3 will serve as thresholds for flow table state detection: when the measured RTT is around T_1, we can infer that there is corresponding flow entry in the flow table; when the measured RTT is around T_2, we can infer that there is no corresponding flow entry in the flow table and the flow table is not full; when the measured RTT is around T_3, we can infer that there is no corresponding flow entry in the flow table and the flow table is full.

We model the SDN/OpenFlow network as a black box and observe its response (RTT) to different input (network packets), then we use the response to estimate the flow table state and flow entry state and perform further inference. The whole process comes in three steps.

Firstly, we send probing packets into the network to trigger the interaction. As there is still no mature routing aggregation algorithm or hierarchical routing rule solution, current SDN/OpenFlow switches typically use exact match rules. That means if we send n packets with different faked metainformation like src_ip and dst_ip, there will be n newly generated flow entries inserted into the flow table. If we send excessive probing packets in a short period of time, the flow table will overflow and then the interaction process will be triggered. Secondly, we measure RTTs of the responded packets and infer the flow table state and flow entry state. Thirdly, we use observed flow table states and flow table states as controlling signals in our inference algorithm and perform flow table capacity inference.

Having to achieve a hit rate as high as possible in a rather limited space, flow table serves like a "cache" in operating systems and web proxy servers. In this paper we choose FIFO and LRU because they are common and popular [13].

3. Inference Algorithm

The logical structure of our inference algorithm is shown in Figure 3. The inference algorithm consists of two main part: flow table state detection and flow table state control. For flow table state detection, we perform RTT measurement to classify the different states of flow table and specific flow entry. For flow table state control, we generate specific sequence of attacking network packets to manipulate the state of flow entries. For different flow table replacement algorithms, the relation between network traffic sequence and flow entry state will be different, so we will have different network traffic generation strategy for different flow table replacement algorithms like FIFO and LRU. We will introduce the inference algorithms for FIFO and LRU, respectively.

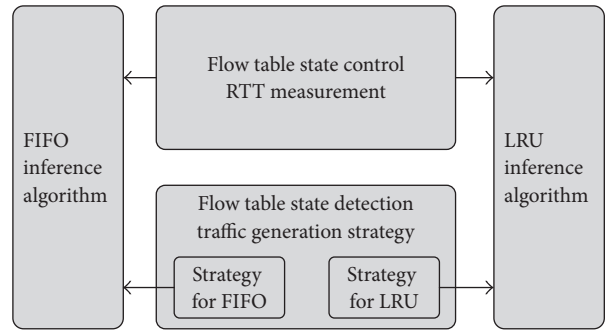


Figure 3: Inference algorithm.

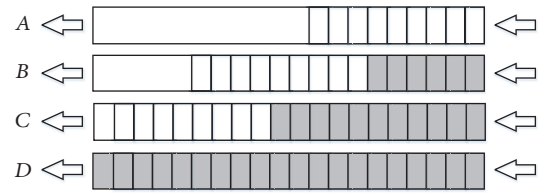


Figure 4: FIFO inference principle.

3.1. FIFO Inference Algorithm. As mentioned in Section 2, the inference process of FIFO algorithm will be as follows: we generate and send a huge amount of probing packets each with a different combination of src_ip, dst_ip, src_mac, and dst_mac, and the newly inserted flow entries matching the generated packets will "push" the other users' flow entries out of the flow table. We can detect if the flow table is full and the existence of our flow entries. Combined with the number of inserted flow entries we recorded, we can infer the flow table capacity and flow table usage. The process of flow table state transformation is shown in Figure 4.

We use F_{our} to represent the number of our inserted flow entries and use F_{other} to represent the number of flow entries from other users in the flow table. Both F_{our} and F_{other} are functions of time. We use T_A, T_B, T_C, and T_D to represent four time points corresponding to four subfigures, respectively, and use C to represent the flow table capacity.

Figure 4 (A) shows the flow table and the flow entries it contains just before the experiment starts. The rectangle items represent the flow entries from other users sharing the OpenFlow switch. The current number of other users' flow entries can be expressed as $F_{\text{other}}(T_A)$.

Figure 4 (B) illustrates the time when we start to send generated packets, inserting new flow entries into the flow table. The grey rectangles represent the flow entries inserted by us. As we can see, our flow entries keep pushing other users' flow entries to the front of the FIFO queue. During the experiment, we should keep a record of the generated packets, including their attributes and serial numbers.

Require:
(1) Packet-Sending Function: *SendPacket*();
(2) List of IP: *IP*;
Ensure:
(3) The flow table capacity: F_{capacity};
(4) The number of other users' flow entries: F_{other};
(5) $F_{\text{capacity}} \leftarrow 0$
(6) $F_{\text{other}} \leftarrow 0$
(7) $N \leftarrow 0$
(8) $N_1 \leftarrow 0$
(9) $N_2 \leftarrow 0$
(10) **while** $N <$ length(IP) **do**
(11) $ip \leftarrow IP[N]$
(12) SendPacket(ip)
(13) $N \leftarrow N + 1$
(14) **if** Flow table is full **then**
(15) $N_1 \leftarrow N$
(16) continue
(17) **end if**
(18) **if** One of our flow entries is deleted **then**
(19) $N_2 \leftarrow N$
(20) break
(21) **end if**
(22) **end while**
(23) $F_{\text{capacity}} \leftarrow N_2$
(24) $F_{\text{other}} \leftarrow N_2 - N_1$
(25) **return** F_{capacity}, F_{other}

Algorithm 1: FIFO inference algorithm.

Figure 4 (C) shows the time when we detect the flow table is full. At this point of time, flow entries from us and other users add up to fill the whole flow table precisely. We have

$$F_{\text{our}}(T_C) + F_{\text{other}}(T_C) = C. \tag{1}$$

Figure 4 (D) shows the time when we detect that one of our inserted flow entries has been deleted. That means the flow table is now full of our flow entries, without any flow entries from other users. We have

$$F_{\text{our}}(T_D) = C. \tag{2}$$

Combine the two equations above; we have

$$\begin{aligned} F_{\text{other}}(T_A) = F_{\text{other}}(T_C) &= C - F_{\text{our}}(T_C) \\ &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C). \end{aligned} \tag{3}$$

According to the analysis above, we describe the inference process for FIFO algorithm as shown in Algorithm 1.

The main error of the inference comes from the flow entries inserted by other users when our insertion is in progress. We assume that our flow entry insertion speed is fast enough so that, during the period of experiment, the newly inserted flow entries are all from us. But that is not always the truth. Ignoring the possible flow entries inserted by other users will make our inference result smaller than the actual value.

Considering the flow entries inserted by other users, the actual equations are listed below.

When we detect the flow table is full, if we use $E(A, B)$ to represent the number of just inserted flow entries from other users from time point A to time point B, the equation becomes

$$F_{\text{our}}(T_C) + F_{\text{other}}(T_C) + E(T_A, T_C) = C. \tag{4}$$

And when we detect that one of our inserted flow entries is deleted, the equation becomes

$$F_{\text{our}}(T_D) + E(T_A, T_C) + E(T_C, T_D) = C. \tag{5}$$

Combine the two equations above; we have

$$F_{\text{other}}(T_C) = F_{\text{our}}(T_D) - F_{\text{our}}(T_C) + E(T_C, T_D). \tag{6}$$

So the actual equation considering flow entry insertions during inference should be

$$\begin{aligned} C &= F_{\text{our}}(T_D) + E(T_A, T_C) + E(T_C, T_D) \\ F_{\text{other}}(T_C) &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C) + E(T_C, T_D). \end{aligned} \tag{7}$$

Compared with our former equation ignoring flow entry insertions,

$$\begin{aligned} C &= F_{\text{our}}(T_D) \\ F_{\text{other}}(T_C) &= F_{\text{our}}(T_D) - F_{\text{our}}(T_C). \end{aligned} \tag{8}$$

We can see that the inferred flow table usage F_{other} and the inferred flow table capacity F_{capacity} will both be smaller than the actual value.

3.2. LRU Inference Algorithm. The experiment principle of LRU algorithm has something in common with that of FIFO algorithm, because under these two circumstances we can both keep our flow entries stay in the back of the cache queue using certain operations. However, there are still differences lies in the flow entry maintaining process.

The nature of FIFO algorithm ensures that the position of the flow entries only depends on the time they are inserted. The earlier inserted flow entries are sure to be nearer to the front of the cache queue compared with the later inserted flow entries. But in LRU algorithm, the positions of the flow entries depend not only on the time they are inserted, but also on the last time they are accessed. In order to keep our flow entries stay in the back of the cache queue, we need to continuously access the previously inserted flow entries.

During the maintain process, every time we insert a new flow entry, we need to access all previously inserted flow entries for one time to "lift" them to the back of the cache queue. The access history may be like $\{P_1\}, \{P_1, P_2\}, \{P_1, P_2, P_3\}, \{P_1, P_2, P_3, P_4\}, \ldots$, and we call it a "rolling" maintaining process. The maintaining algorithm is shown in Algorithm 2. According to the analysis above, we describe the inference process for LRU in Algorithm 3.

The feasibility and error analysis of LRU algorithm is similar to that of FIFO algorithm. The inferred flow table usage F_{other} and the inferred flow table capacity F_{capacity} will both be smaller than the actual value because of ignoring the flow entries inserted by other users during the experiment.

```
Require
 (1) Packet-Sending Function: SendPacket();
 (2) List of Inserted IP: IP_inserted;
 (3) function RollingPacketSender(IP_inserted)
 (4)     i ← 1
 (5)     while i < length(IP_inserted) do
 (6)         for j ← 0; j < i; j + + do
 (7)             ip ← IP_inserted[j]
 (8)             SendPacket(ip)
 (9)         end for
(10)         i ← i + 1
(11)     end while
(12) end function
```

Algorithm 2: Rolling maintaining algorithm.

```
Require:
 (1) Packet-Sending Function: SendPacket();
 (2) List of IP: IP;
Ensure:
 (3) The flow table capacity: F_capacity;
 (4) The number of other users' flow entries: F_other;
 (5) F_capacity ← 0
 (6) F_other ← 0
 (7) N ← 0
 (8) N_1 ← 0
 (9) N_2 ← 0
(10) IP_inserted ← []
(11) while N < length(IP) do
(12)     ip ← IP[N]
(13)     IP_inserted ← IP_inserted + ip
(14)     RollingPacketSender(IP_inserted)
(15)     N ← N + 1
(16)     if Flow table is full then
(17)         N_1 ← N
(18)         continue
(19)     end if
(20)     if One of our flow entries is deleted then
(21)         N_2 ← N
(22)         break
(23)     end if
(24) end while
(25) F_capacity ← N_2
(26) F_other ← N_2 − N_1
(27) return F_capacity, F_other
```

Algorithm 3: LRU inference algorithm.

4. Evaluation

4.1. Implementation. The emulation environment of our experiment consists of three parts: a network prototyping system used to emulate host and switch, a network controller, and our inference attack toolkit.

We choose Mininet [14] as the network prototyping system because it encapsulates host and switch emulation and thus easy to use. Our emulated network prototype for evaluation uses a star topology, consisting of 20 hosts connected to a single OpenFlow switch. We build FIFO and LRU controller applications using Python on the basis of POX [15] OpenFlow controller. As for the inference attack toolkit, we use libnet [16] to generate probing packets, and libpcap [17] to capture replied packets. To simulate the background traffic in real network, we built a SDN testbed using Mininet and POX. On the SDN testbed, we performed a series of basic SDN operations. These operations include building a customized SDN network topology, setting up the link between SDN switches and performing the ping test between all SDN nodes. We captured the network traffic generated during these operations and use them as the background network traffic sample.

4.2. RTT Measurement. As we have mentioned in Section 2, the difference between traditional network and SDN/OpenFlow network in handling previously unseen packets gives us a possible indicator of the flow table state and the flow entry living state, RTT. When there is not corresponding flow entry existing in the flow table, the RTT of a packet will significantly increase due to the interactions between controller and switch in order to acquire new flow entries. That is the case when there is still space in the flow table. Once the flow table is full, the RTT of a packet will further increase as a result of extra flow table replacement operations. To prove the effectiveness of using RTT as the flow table state and flow entry state indicator, we measured packet RTTs corresponding to different flow table state and flow entry state.

Figure 5 gives the RTT measurement result showing the difference. The points with different symbols represent the total 300 times of RTT measurements we have conducted, 100 times of measurement for each combination of flow table state and flow entry state. The square points stand for RTTs when flow entry exists in flow table. The circle points and triangle points both stand for RTTs when flow entry does not exist in flow table; the circle points are measured when the flow table is full, and the triangle points are measured when the flow table is not full.

As can be seen from the figure, when flow entry exists in flow table, the packet RTTs are highly concentrated in the

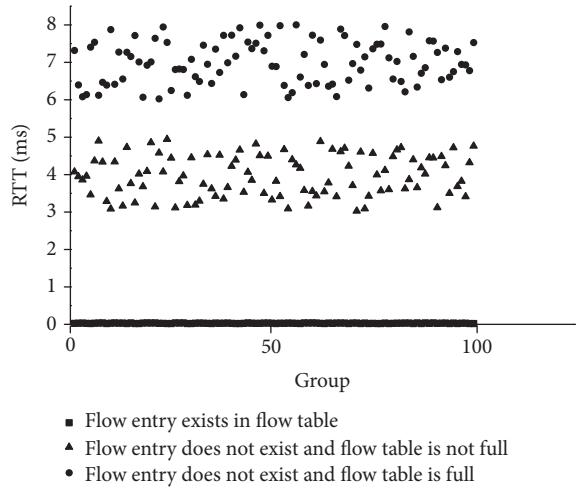


Figure 5: RTT measurement.

Table 1: Default timeout values.

Controller	Hard_timeout	Idle_timeout
Ryu	0	0
Beacon	0	5 s
Floodlight	0	5 s
NOX	0	5 s
POX	30 s	10 s
Trema	0	60 s
Maestro	180 s	30 s

range of 0.2~0.3 ms; when flow entry does not exist in flow table and flow table is not full, the packet RTTs will increase to about 3~5 ms; when flow entry does not exist in flow table and flow table is full, the packet RTTs will be the highest, ranging from 6 ms to 8 ms. These three groups of RTTs all distribute intensively in a small range without overlapping other groups, showing the excellent discrimination of using RTT as a flow table state and flow entry state indicator.

4.3. Timeout

4.3.1. Default Timeout Values. According to our previous analysis, the feasibility of our inference attack depends on whether we can generate enough flow entries to fulfill the flow table within a single timeout cycle. That means we must have the ability to generate as many flow entries as the flow entry can hold during a timeout period. So we analyze several popular open-source controllers and search for their default timeout values in the built-in applications. The result is presented in Table 1. The zero values in the table mean the corresponding timeout will not take effect, or in other words the timeout value is "permanent." As can be seen from the table, most available controllers have timeout values in the range of 5 s to 30 s.

If we take the flow table capacity of 2000 flow entries as an example, the minimum packet generating speed required will be 2000/5 = 400 packets per second, while libnet can generate tens of thousand packets per second. So the default timeout values ensure the feasibility of our inference attack.

4.3.2. Timeout Measurement. Though default timeout values of mainstream OpenFlow controllers can be read from their source codes, it is still possible for SDN network administrators to manually change the default timeout values. In order to handle nondefault timeout values and provide basis for adjusting packet generating speed, it is essential to examine the accuracy of passive timeout measurement.

Figure 6 illustrates relative errors (see equation (9)) of hard_timeout and idle_timeout measurement, respectively. We manually modify hard_timeout and idle_timeout values of POX OpenFlow controller to 5 s, 10 s, 15 s, 20 s, 25 s, and 30 s, and then we use timeout measurement algorithm mentioned in Section 2 to measure these timeout values and calculate relative errors:

$$\text{relative\_error} = \frac{\left|\text{value}_{\text{detected}} - \text{value}_{\text{true}}\right|}{\text{value}_{\text{true}}} * 100\%. \quad (9)$$

Every line in Figures 6(a) and 6(b) corresponds to 10 times of repeated measurements conducted under a certain timeout setting from 5 s to 30 s. The margin stays in the range of plus-or-minus 10 percent, showing the effectiveness and high accuracy of our timeout measurement algorithm.

4.4. Flow Table Capacity. Flow capacity is the primary target of our inference attack. It reflects the hardware specification of an OpenFlow switch. Figure 7 illustrates the flow table capacity measurement result when controller adopts FIFO replacement algorithm. We manually limited the switch flow table capacity to 10 different values from 100 flow entries to 1000 flow entries and used our framework to perform the inference.

The dark bars represent the manually set flow table capacities or *real* capacities. The light bars represent the measured flow table capacities. For every manually set flow table capacity, we conduct 10 times of repeated measurements and take their mean value as the final result. From the figure we can see that the measured capacities are quite close to the real capacities, indicating the high accuracy of our inference framework. For example, when the real capacity is 400 flow entries, our measured capacity is 408 flow entries with an error of only 8 flow entries. As the real capacity grows, the packet generating speed required becomes faster, placing higher requirements on packet sending, receiving synchronization and accurate timing. But our inference algorithm shows unbelievable stability and accuracy: when the real capacity is 1000 flow entries, our measured capacity is 973 flow entries with an error of just 27 flow entries.

Like Figure 7, Figure 8 also illustrates the flow table capacity measurement results, with the only difference of being performed under LRU replacement algorithm instead of FIFO.

According to our previous analysis, the inference principle of LRU replacement algorithm is more complex because of the unavoidable mixed nature of flow entries in the flow table and the rolling maintaining process. But our inference

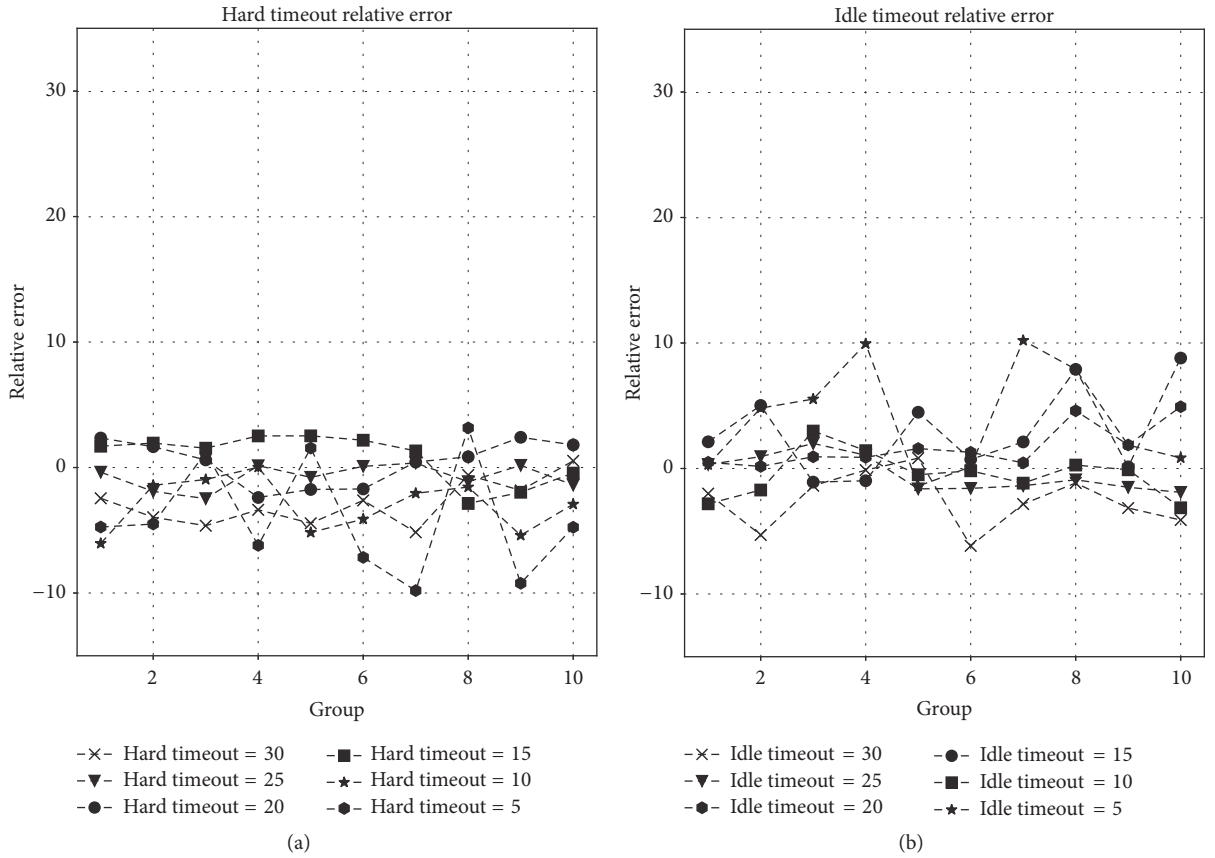


Figure 6: Timeout relative error.

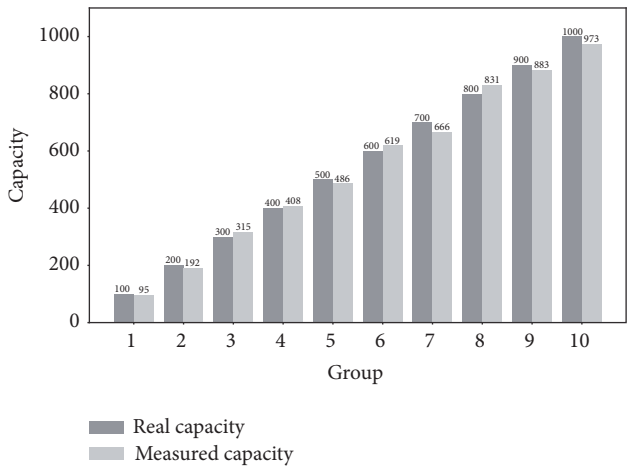


Figure 7: FIFO flow table capacity.

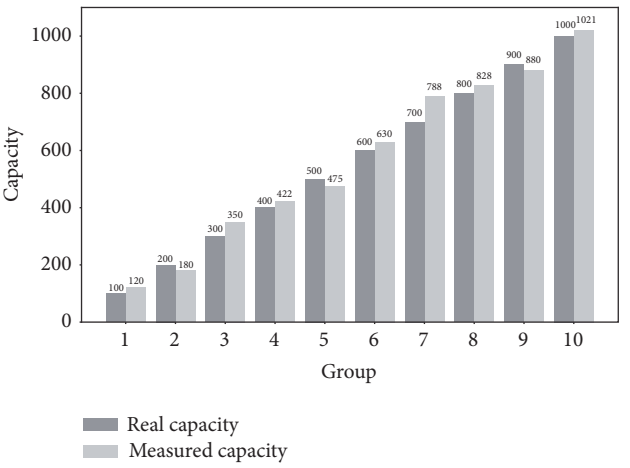


Figure 8: LRU flow table capacity.

framework still shows high accuracy and reliability. Even when the real flow table capacities are set to be rather large values like 900 and 1000, the errors of our measure capacities are just around 20 flow entries.

Only illustrating the mean value of measured flow table capacities may not be enough: the mean value may be the result of error compensations and hide the detailed measurement errors of every separate experiment. So in Figure 9 we illustrate the relative error of every single flow table capacity measurement.

We choose 5 groups of different flow table capacities from 200 flow entries to 1000 flow entries and perform 10 times of measurements under every single flow table capacity value. Figure 9(a) stands for relative error of flow table capacity measurements conducted under FIFO replacement

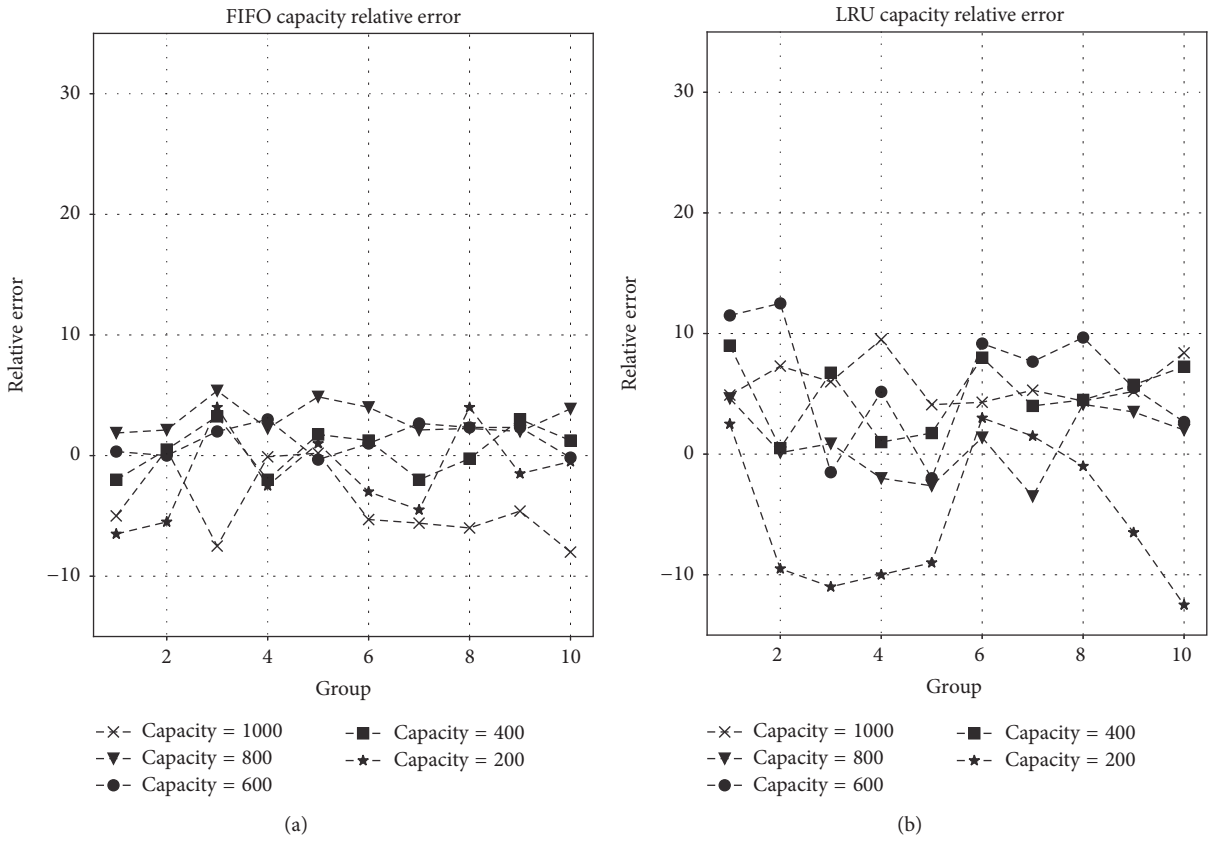


Figure 9: Flow table capacity relative error.

algorithm, showing that the margin is no larger than plus-or-minus 10 percent. Figure 9(b) stands for relative error of flow table capacity measurements conducted under LRU replacement algorithm. Due to the more complex inference principle and the rolling maintaining process, the margin becomes larger but still has not exceeded 15 percent even in the worst case.

The above inference attacks are performed without any background network traffic. When performing inference attack in real networks, the impact of background network traffic cannot be ignored. So it is necessary to examine the efficiency of our inference algorithm under these circumstances.

In this evaluation, we choose the background network traffic dataset from a SDN testbed. Figures 10 and 11 have the same experiment setting with Figures 7 and 8, with the only difference of replaying background traffic captured from SDN testbed during the inference attack process. Even with the impact of background traffic, our inference algorithm still shows high accuracy.

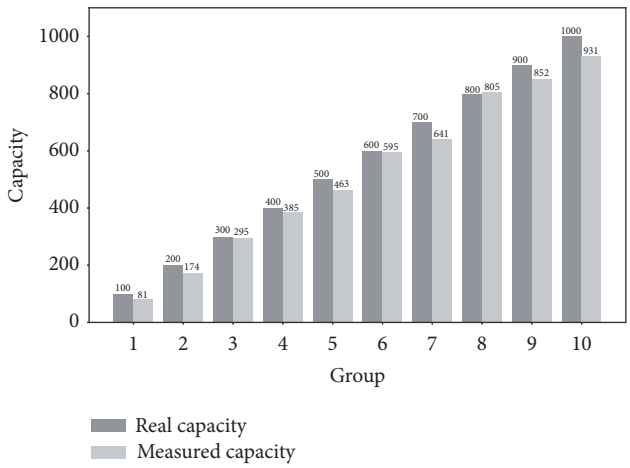


Figure 10: FIFO flow table capacity with testbed background traffic.

4.5. Flow Table Usage. In this section we evaluated our framework's efficiency of inferring the number of flow entries from other users sharing the same flow table or the flow table usage. Flow table usage is our secondary inference target, and it reflects the network resource consuming condition of other tenants in the same SDN network. Figures 12 and 13 illustrate the flow table usage measurement results conducted under FIFO and LRU replacement algorithm, respectively.

Again we manually set 10 different flow table usage values from 100 to 1000 flow entries by manually generating and inserting corresponding number of flow entries into the flow table beforehand. Then we use our inference algorithm to infer the flow table usage and take mean values of every 10 times of measurements as the final results. The errors of all these measurements show the high accuracy, stability and reliability of our inference algorithm.

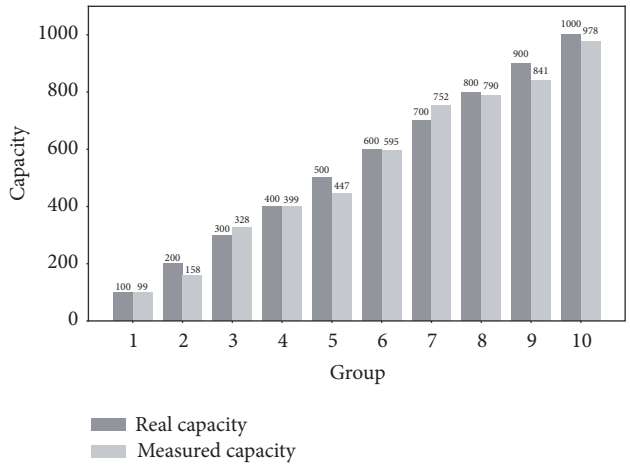


Figure 11: LRU flow table capacity with testbed background traffic.

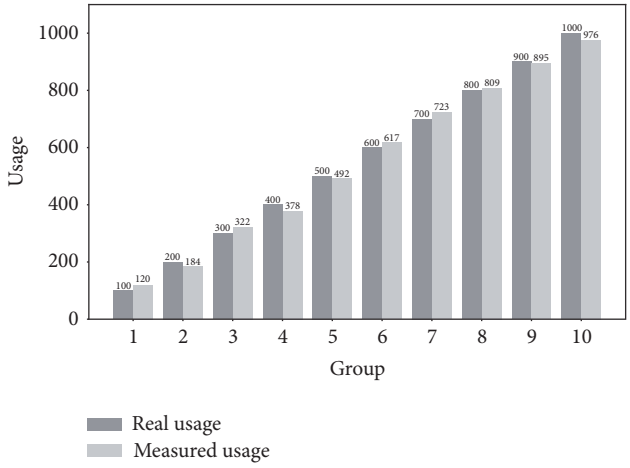


Figure 12: FIFO flow table usage.

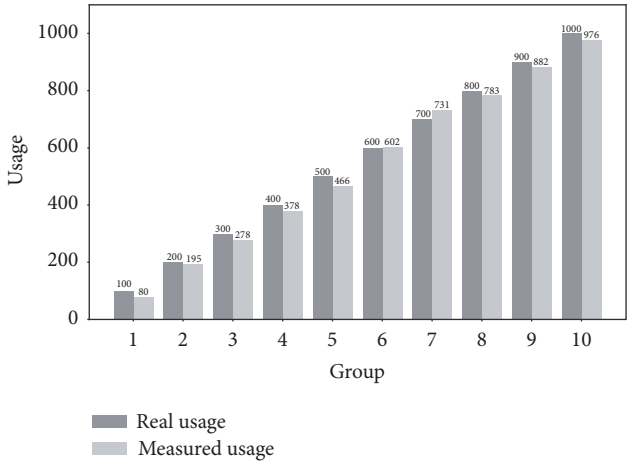


Figure 13: LRU flow table usage.

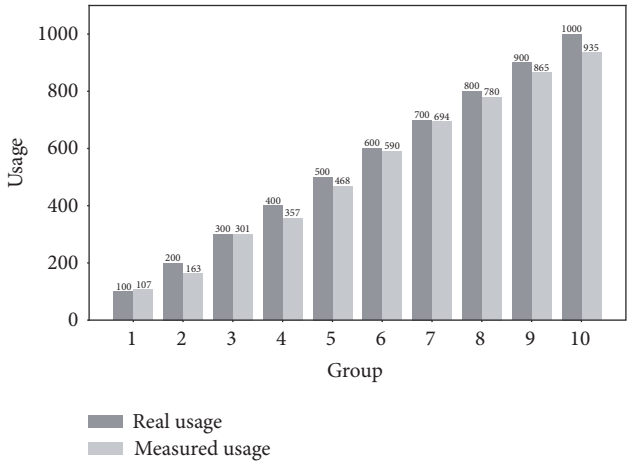


Figure 14: FIFO flow table usage with testbed background traffic.

We also conducted the flow table usage inference attack with background traffic. Experiment results with testbed background traffic are shown in Figures 14 and 15. Our inference algorithm can smoothly handle the impact of background traffic, which ensures the stability and robustness demonstrated in the experiment results.

The relative errors are shown in Figure 16. We emulate 5 groups of different flow table usage values and conducted 10 times of flow table usage inference for every single value. For both FIFO and LRU replacement algorithm, the relative errors of flow table usage inference stay in a quite small range. The results prove that our algorithm can infer other tenants' flow table usage condition in high accuracy.

5. Defense

From the previous sections we can conclude that the inference attack is rooted in the flow table overflow. To defend that kind of inference attack, we have to prevent flow table overflow in two aspects: one aspect is to compress the flow entries to save flow table space, and the other one is to implement a larger flow table to store more flow entries.

5.1. Routing Aggregation. Routing aggregation is to combine multiple entries in the flow table without changing the next hops for packet forwarding. This approach is particularly appealing because it can be done by a software upgrade at the OpenFlow switch and its impact is limited within that switch. Routing aggregation has already been used in traditional networks, but it has not been deployed in SDN/OpenFlow networks. To fully utilize the flexibility of SDN/OpenFlow network under certain scenarios like load balancing, we proposed a global routing schedule using packing optimization algorithms.

Traditional routing aggregation algorithms [18] can be used to compress the flow table, but their effectiveness cannot be ensured. If the matching fields and next hops are dispersed enough, chances are that we may not be able to perform any routing aggregation because we cannot find flow entries sharing common matching fields and next hops. This is often

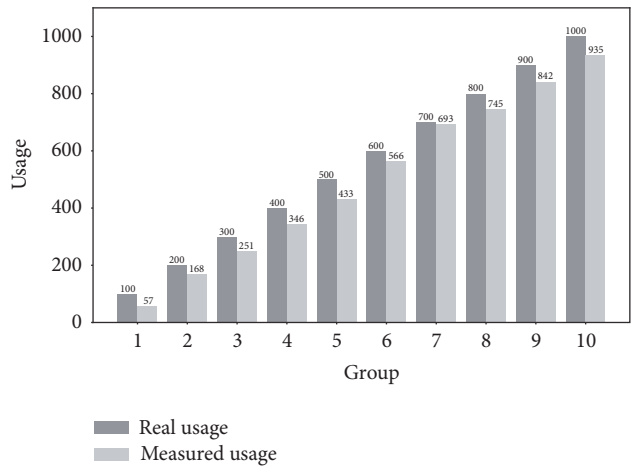


Figure 15: LRU flow table usage with testbed background traffic.

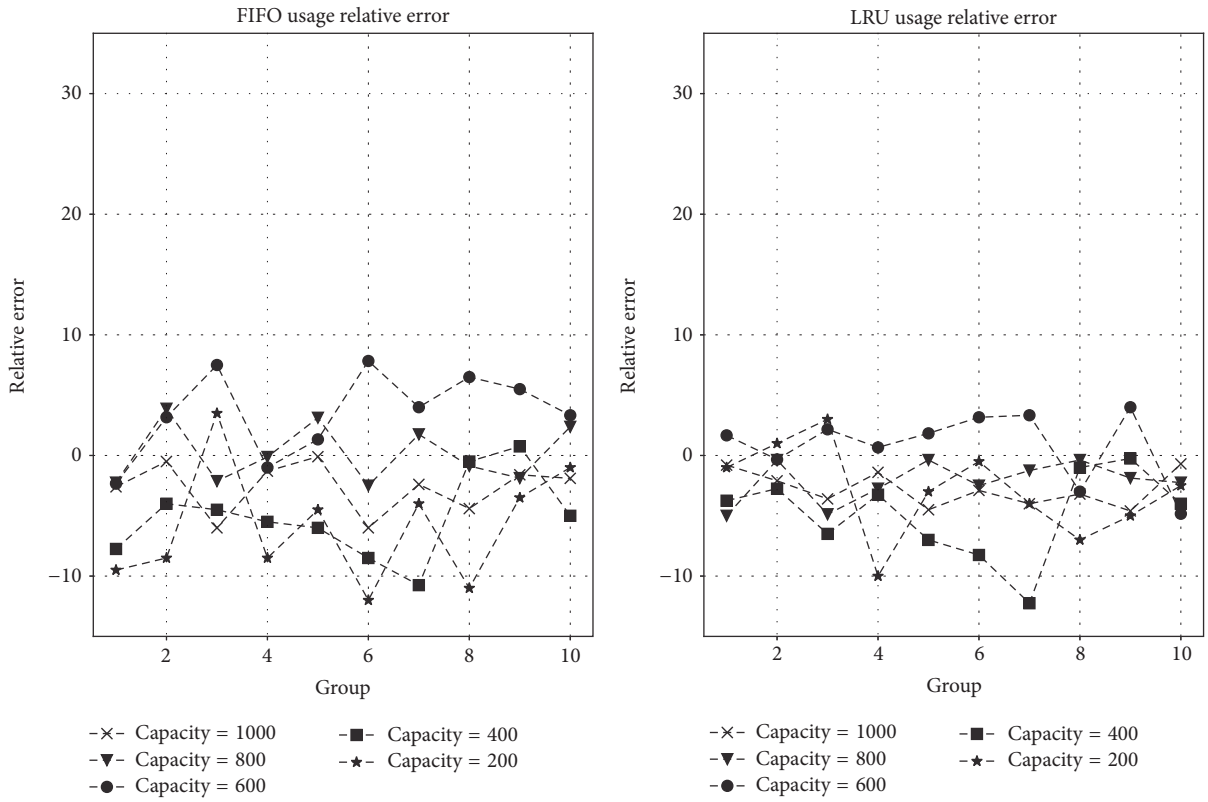


Figure 16: Flow table usage relative error.

the case when dealing with web traffic, for example, load balancing services. For that reason, we introduced an extra stage of routing aggregation: global routing schedule optimization. First we model this routing aggregation problem as a packing optimization problem and solve it, then we perform global routing reschedule by rewriting the flow entries according to the optimization result, and finally we perform another time of traditional routing aggregation on these new flow entries. After the global routing reschedule, there will be much more aggregatable flow entries, so the effectiveness of routing aggregation is ensured.

5.2. Multilevel Flow Table Architecture. It is important to note that routing aggregation is not a replacement for the long-term architectural solutions because it does not address the root causes of the flow table scalability problem and the following inference attack. To eliminate the inference attack vulnerability, a flow table architecture with larger capacity is

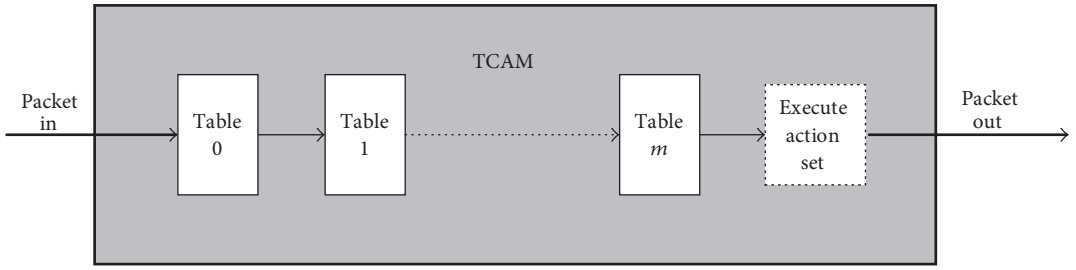


Figure 17: Single-level flow table architecture.

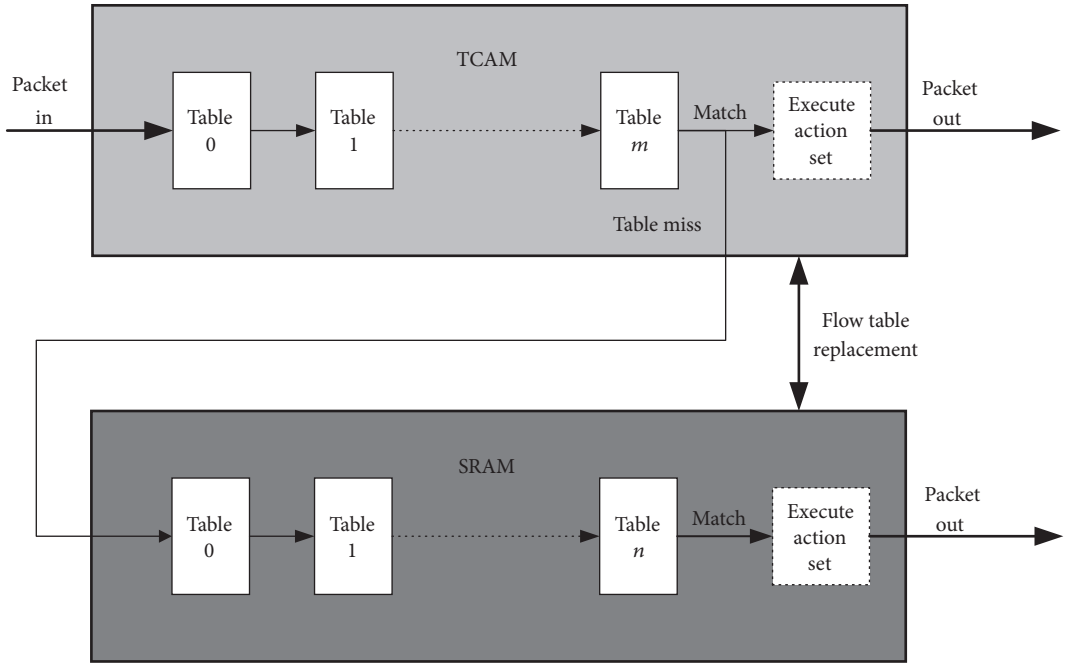


Figure 18: Multilevel flow table architecture.

required, which can be achieved through multilevel flow table consisting of both TCAM and SRAM.

The original single-level flow table architecture is shown in Figure 17. In this architecture, the flow table is completely implemented using TCAM. An input packet will traverse from table 0 to table m and add corresponding actions to the action set. Then all actions in the action set are executed and the packet is forwarded according to these actions.

Our proposed multilevel flow table architecture is shown in Figure 18. Besides flow table implemented using TCAM, we add another flow table implemented using SRAM, which is cheaper and can provide larger flow table space. Under this multilevel flow table architecture, the packet processing pipeline will be different: first an input packet will find matching flow entries in TCAM flow table, just like in the original single-level flow table architecture. If there is a match, the packet will execute the corresponding actions and get forwarded. If there is no match, the packet will continue its lookup in the SRAM flow table. If there is a match in the SRAM flow table, the packet can then be forwarded; otherwise it will be sent to the controller.

From the process above, we can see that if the capacity of TCAM flow table is m, the capacity of SRAM flow table is n, and then the multilevel flow table will have a capacity of $m + n$. Actually n is far more larger than m, so this approach can greatly increase the flow table capacity, thus preventing flow table overflow.

6. Discussion

SDN/OpenFlow has become a competitive solution for next-generation network and is being more and more widely used in modern datacenters. But considering its key role as the fundamental infrastructure, we have to admit that the security issues of SDN/OpenFlow have not been explored to a large extent. Particularly, the flow table capacity of SDN/OpenFlow switch is only considered as a vulnerable part for DDoS and flooding attacks in published researches. But according to our analysis in previous sections, the flow table capacity can lead to potential inference attack if combined with reasonable assumptions and RTT measurements.

Firstly, we found in Section 2 that exact match flow entries as well as the lack of route aggregation would consume a lot

of flow table space, making it impossible to process millions of flows per seconding using SDN/OpenFlow. Secondly, we found in Section 4 that assigning the decision making job of flow table replacement to the controller would lead to significant network performance decrease, which had to be changed in time. Thirdly, there is currently no mature attack detection mechanism for SDN/OpenFlow network, so it is quite easy for criminals to exploit system vulnerabilities or invoke DDoS attacks.

The inference method proposed in this paper just uses some basic elements and parameters of OpenFlow, such as idle timeout and hard timeout, which are significant for the implementation of SDN. These features will not be removed except very huge changes made. On the other hand, although some security frameworks [3] were proposed to detect the malicious insertion of flow rules, attackers can also bypass the detection by some well-designed insertion strategies.

All these security issues call for improvements to current OpenFlow switch and flow table design. The improvements should at least contain the following aspects: (1) New OpenFlow switch architecture, like embedding local caches in the switch or implementing multilevel flow table to achieve a much larger flow table capacity. With larger flow table capacity, the switch will not have to query the controller for flow entries, which will reduce the interaction latency to a large extent. (2) New flow table maintaining mechanism, like transferring the flow entry deleting workload from controller to switch. Switch itself can decide which flow entry to delete and then sync state with controller, and during the flow entry deleting process, the controller's intervention is not needed. In the widely used OpenFlow Switch Specification 1.4.0 [19], this mechanism has been added as an optional feature, but without any mature implementation so far. (3) Routing aggregation. Routing aggregation can match a group of flows using one flow entry, which will reduce the flow table consuming significantly compared with exact match. (4) Inference attack detection. Administrators can develop inference attack detecting applications and then perform defenses like portspeed limiting or network address validation.

From the discussion above, we can see that there is still a long road to go before SDN/OpenFlow becomes a truly mature and reliable network paradigm. There are still urgent and severe issues to solve, which have been neglected in the past. Only by solving these security issues and architectural vulnerabilities can SDN/OpenFlow be widely deployed in real-world commercial datacenters and fully demonstrate its revolutionary flexibility and intelligence.

7. Related Work

The inference attack proposed in this paper is motivated by the limited flow table capacity of SDN/OpenFlow switches. The flow table capacity issue has been presented in many previous works like [20–24]. They all point out the limitation of switch flow table memory and potential scalability and security issue. However, these works do not give further analysis on the inference attack and information leakage caused by the limited flow table capacity.

Klöti et al. [25] present potentially problematic issues in SDN/OpenFlow including information disclosure through timing analysis. However, this information disclosure requires disclosing existing flows with side channel attack, which is hard to perform in real world. Compared with their approach, our inference attack is self-contained and requires no prior knowledge.

Gong et al. [26] present a kind of inference attack using RTT measurement to infer which website the victim is browsing. They recover victims' network traffic patterns based on the queuing side channel happened at the Internet router. However, the scenario of their work is in the public Internet, while our approach focuses on SDN/OpenFlow infrastructures in cloud computing network. Compared with public Internet and website inference, the inference attack and information leakage in modern data centers are more sensitive and valuable.

Shin and Gu [27] demonstrate a novel attack targeting at SDN networks. This attack includes fingerprinting SDN networks and further flooding the data plane flow table by sending specifically crafted fake flow requests in high speed. In the fingerprinting phase, header field change scanning is used to collect the different response time (RTT) for new flow and existing flow. The fingerprinting result is then analyzed to estimate if the target network used SDN technology. The RTT measurement and analysis they used in fingerprinting are similar to our approach. But they just perform DoS attacks to the SDN network, without performing any further information leakage or network parameter inference.

As for flow table overflow defending strategy, Shelly et al. [28] and Katta et al. [29] introduce flow entry caching mechanism into SDN/Openflow network by inserting a transparent intermediate layer between controller and switch. Yan et al. [9] use CAB to generate wildcard flow entries dynamically and reactively to handle bursting network traffic. Kannan and Banerjee [30] present a flow entry compaction algorithm to save TCAM flow table space. This algorithm uses flow entry tags instead of matching fields as forwarding rules. Kim et al. [31] develop a new flow entry management scheme to reduce the controller overhead.

8. Conclusion

In this paper, we have explored the structure of SDN/OpenFlow network and some of the possible security issues it brings. After our detailed analysis of the SDN/OpenFlow network, we proposed a novel inference attack model targeting at the SDN/OpenFlow network, which is the first proposed inference attack model of this kind in the SDN/OpenFlow area. This inference attack is introduced by the OpenFlow switch, especially by its limited flow table capacity. The inference attack can be done in a completely passive way, making it hard to detect and defend. We also implemented the inference attack framework and examined the efficiency and accuracy of it using network traffic data from different sources. The simulation results show that the inference attack framework can infer the network parameter (flow table capacity and flow table usage) with an accuracy of up to 80% or higher. We also proposed two possible defense strategies for the discovered

vulnerability, including routing aggregation algorithm and multilevel flow table architecture.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The research presented in this paper is supported in part by the National Key Research and Development Program of China (no. 2016YFB0800100), the Fund of China National Aeronautical Radio Electronics Research Institute (PM-12210-2016-001), the National Natural Science Foundation (61572397, U1766215, U1736205, 61502383, 61672425, and 61702407), and State Grid Corporation of China (DZ71-16-030).

References

[1] W. Li, W. Meng, and L. F. Kwok, "A survey on OpenFlow-based Software Defined Networks: Security challenges and countermeasures," *Journal of Network and Computer Applications*, vol. 68, pp. 126–139, 2016.

[2] M. Brooks and B. Yang, "A Man-in-the-Middle attack against OpenDayLight SDN controller," in *Proceedings of the 4th Annual Conference on Research in Information Technology*, Association for Computing Machinery, Chicago, IL, USA, September 2015.

[3] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks, HotSDN 2012*, pp. 121–126, Association for Computing Machinery, Helsinki, Finland, August 2012.

[4] N. Gude, T. Koponen, and J. Pettit, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[5] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM SIGCOMM 2012*, pp. 467–472, Finland, August 2012.

[6] G. Zhao, L. Huang, Z. Yu, H. Xu, and P. Wang, "On the effect of flow table size and controller capacity on SDN network throughput," in *Proceedings of the 2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, Paris, France, May 2017.

[7] A. Roy, H. Zengy, J. Baggay, G. Porter, and A. C. Snoeren, "Inside the social network's (Datacenter) network," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pp. 123–137, UK, August 2015.

[8] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 151–156, Chicago, Illinois, USA, August 2014.

[9] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A reactive wildcard rule caching system for software-defined networks," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 163–168, Chicago, Illinois, USA, August 2014.

[10] Katta Naga, Jennifer Rexford, and David Walker, "Infinite cacheflow in software-defined networks," Princeton School of Engineering and Applied Science, 2013.

[11] Yanwen Wang, Hainan Chen, Hainan Chen, and Lei Shu, "An energy-efficient SDN based sleep scheduling algorithm for WSNs," *Journal of Network and Computer Applications*, vol. 59, pp. 39–45, 2016, http://dx.doi.org/10.1016/j.jnca.2015.05.002.

[12] S.-N. Yang, S.-W. Ho, Y.-B. Lin, and C.-H. Gan, "A multi-RAT bandwidth aggregation mechanism with software-defined networking," *Journal of Network and Computer Applications*, vol. 61, pp. 189–198, 2016, http://dx.doi.org/10.1016/j.jnca.2015.11.003.

[13] M. Dehghan, L. Massoulie, D. Towsley, D. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications, IEEE INFOCOM 2016*, San Francisco, CA, USA, April 2016.

[14] "Mininet" http://mininet.org/.

[15] "POX Controller" http://www.noxrepo.org/pox/about-pox/.

[16] "libnet-dev" https://github.com/sam-github/libnet.

[17] "libpcap" http://www.tcpdump.org/.

[18] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *Proceedings of IEEE INFOCOM 2010*, Institute of Electrical and Electronics Engineers, San Diego, CA, USA, March 2010.

[19] "OpenFlow Switch Specification 1.4.0" https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[20] S. Sezer, S. Scott-Hayward, P. Chouhan et al., "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[21] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[22] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proceedings of the 2013 Workshop on Software Defined Networks for Future Networks and Services, SDN4FNS 2013*, Trento, Italy, November 2013.

[23] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani, "Securing software defined networks: Taxonomy, requirements, and open issues," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 36–44, 2015.

[24] M. Tsugawa, A. Matsunaga, and J. A. B. Fortes, "Cloud computing security: What changes with software-defined networking?" *Secure Cloud Computing*, pp. 77–93, 2014.

[25] R. Klöti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proceedings of the 2013 21st IEEE International Conference on Network Protocols, ICNP 2013*, Institute of Electrical and Electronics Engineers, Goettingen, Germany, October 2013.

[26] X. Gong, N. Borisov, N. Kiyavash, and N. Schear, "Website Detection Using Remote Traffic Analysis," in *Privacy Enhancing Technologies*, vol. 7384 of *Lecture Notes in Computer Science*, pp. 58–78, Springer, Berlin, Heidelberg, 2012.

[27] S. Shin and G. Gu, "Attacking software-defined networks: a first feasibility study," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*, pp. 165-166, Association for Computing Machinery, Hong Kong, China, August 2013.

[28] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 663–668, 2014.

[29] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in software-defined networks," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 175–180, Association for Computing Machinery, Chicago, Illinois, USA, August 2014.

[30] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proceedings of the 14th International Conference on Distributed Computing and Networking*, vol. 7730 of *Lecture Notes in Computer Science*, pp. 439–444, Springer, Mumbai, India, 2013.

[31] E.-D. Kim, S.-I. Lee, Y. Choi, M.-K. Shin, and H.-J. Kim, "A flow entry management scheme for reducing controller overhead," in *Proceedings of the 16th International Conference on Advanced Communication Technology: Content Centric Network Innovation!, ICACT 2014*, pp. 754–757, Institute of Electrical and Electronics Engineers, Pyeongchang, South Korea, February 2014.

Hindawi
Security and Communication Networks
Volume 2017, Article ID 3659167, 11 pages
https://doi.org/10.1155/2017/3659167

Research Article

CHAOS: An SDN-Based Moving Target Defense System

Yuan Shi,[1] Huanguo Zhang,[1] Juan Wang,[1] Feng Xiao,[1] Jianwei Huang,[1] Daochen Zha,[1] Hongxin Hu,[2] Fei Yan,[1] and Bo Zhao[1]

[1]*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, Computer School of Wuhan University, Wuhan, China*
[2]*Division of Computer Science, School of Computing, Clemson University, Clemson, SC 29634, USA*

Correspondence should be addressed to Juan Wang; jwang@whu.edu.cn

Received 2 August 2017; Accepted 11 September 2017; Published 16 October 2017

Academic Editor: Zhiping Cai

Moving target defense (MTD) has provided a dynamic and proactive network defense to reduce or move the attack surface that is available for exploitation. However, traditional network is difficult to realize dynamic and active security defense effectively and comprehensively. Software-defined networking (SDN) points out a brand-new path for building dynamic and proactive defense system. In this paper, we propose CHAOS, an SDN-based MTD system. Utilizing the programmability and flexibility of SDN, CHAOS obfuscates the attack surface including host mutation obfuscation, ports obfuscation, and obfuscation based on decoy servers, thereby enhancing the unpredictability of the networking environment. We propose the Chaos Tower Obfuscation (CTO) method, which uses the Chaos Tower Structure (CTS) to depict the hierarchy of all the hosts in an intranet and define expected connection and unexpected connection. Moreover, we develop fast CTO algorithms to achieve a different degree of obfuscation for the hosts in each layer. We design and implement CHAOS as an application of SDN controller. Our approach makes it very easy to realize moving target defense in networks. Our experimental results show that a network protected by CHAOS is capable of decreasing the percentage of information disclosure effectively to guarantee the normal flow of traffic.

1. Introduction

Nowadays, the network security issues become increasingly prominent as all kinds of network security events emerge one after another. However, the traditional network security tools cannot effectively defend increasingly complex and intelligent penetration of network intrusion and unknown vulnerability attacks. As usually, adversaries can break through or bypass firewalls and intrusion detection systems (IDS) so that an intranet can be easily compromised. As one of revolutionary technologies, Moving Target Defense (MTD) changes game rules, providing a dynamic and proactive network defense [1–3].

MTD aims at building a dynamically and continually shifting and changing system to increase complexity and cost for attackers, limit the exposure of vulnerabilities and opportunities for attackers, and increase system resiliency [4]. The idea of MTD has been applied to network security, for example, DYNAT [5] and DESIR [6].

The difference between MTD and traditional network tools, such as firewall and IDS, is that the latter will suspend suspicious actions once they break security rules. That makes it easy for adversaries to figure out the deployed network defense mechanism so that they will try to bypass them. However, MTD sends illegible fake information to potential threats to make them spend more time and cost so that they will leave more footprints, making them easier to be exposed.

However, due to its closed and static characteristics, traditional network is difficult to realize dynamic and proactive security defense effectively and comprehensively. As a new type of network security architecture, software-defined networking (SDN) points a brand-new path for building dynamic and proactive defense system [7, 8]. SDN has a couple of benefits. It decouples network control and data planes, enabling network control to become directly programmable [9]. It enables network managers to configure, manage, secure, and optimize network resources very quickly via dynamic and automated SDN programs [10]. Meanwhile,

SDN lets the underlying infrastructure be abstracted from applications and network services [11]. In addition, SDN controllers can provide a global view of the network. The central management of SDN makes networks more intelligent.

Therefore, our goal is to build an SDN-based dynamic network defense system. In order to realize the SDN-based MTD, it has some key challenges to be resolved. Firstly, we should leverage SDN to obfuscate network fingerprinting. Secondly, the moving target defense may make some networks services unavailable, such as database server. The IP address and port number of these services have to be opened to the outside and remain real. If MTD obfuscates these services fully, it will return users with fake IPs and ports, making these services unable to be used. Thirdly, obfuscating network parameters indiscriminately will severely reduce the performance of networks undoubtedly.

Motivated by the aforementioned goals and challenges, we propose CHAOS, a SDN-based MTD system. Utilizing the programmability and flexibility of SDN, CHAOS obfuscates the attack surface including host mutation obfuscation, ports obfuscation, and obfuscation based on decoy servers thereby enhancing the unpredictability of the networking environment. Furthermore, it discriminately obfuscates hosts with different security levels in networks. In CHAOS, we propose the Chaos Tower Obfuscation (CTO) method, which uses the Chaos Tower Structure (CTS) to depict the hierarchy of all the hosts in an intranet and define expected connection and unexpected connection. Moreover, we develop fast CTO algorithms to achieve a different degree of obfuscation for the hosts in each layer. We design and implement CHAOS as an application of SDN controller. Our approach makes it very easy to realize moving target defense in networks.

Furthermore, we evaluate our system and the results show that CHAOS can effectively hide real information of the target hosts from attackers and produce fake responses, which can disrupt an adversary's ability to sniff network traffic effectively. In addition, our tests show that the system has lower cost when compared with a fully obfuscated system, which strengthens its applicability in real networks.

Our contributions can be summarized as follows:

(i) We propose a new SDN-based MTD approach, CHAOS, where a Chaos Tower Structure (CTS) is constructed to represent a hierarchy of all the hosts in the network. Using the CTS, we can determine if a network connection is needed to be obfuscated.

(ii) We present a more unpredictable and flexible obfuscation method named Chaos Tower Obfuscation (CTO) in CHAOS, where the level of obfuscation is decided reasonably. Furthermore, through using host mutation obfuscation, ports obfuscation, and obfuscation based on decoy servers, CHAOS can flexibly forward and modify the packets in a network to obfuscate the attack surface.

(iii) We design and implement CHAOS as an SDN application and evaluate its performance. The results demonstrate that a network protected by CHAOS can decrease the percentage of information disclosure effectively and has a lower cost.

(iv) CHAOS is designed and implemented as an application of SDN controller and works with IDS that lets it very easy to realize moving target defense in networks, so CHAOS not only solves the key issues of building a practical SDN-based MTD system, but also can be used in the real-world systems instead of a theory model.

The remainder of this paper is organized as follows. Section 2 provides some background information relating to our system. Section 3 describes how we design our system. Section 4 shows the details of CHAOS obfuscation methods. Section 5 presents the implementation and evaluation of our system. Section 6 shows some related work. Section 7 concludes this paper.

2. Background and Threat Model

In this section, we first provide an introduction to SDN and its mechanism of asynchronous messaging. Then we introduce a threat model about our system.

2.1. SDN and Its Asynchronous Messaging Mechanism. SDN has emerged as a programmable and centrally controlling architecture providing an agile platform for vendors as well as enterprise users to control and define network.

The SDN controller plays the role of an operating system (OS) for networks [11]. All communications between network applications and network devices have to go through the controller. OpenFlow protocol as the first SDN standards defined the communication protocol between the SDN controller and the forwarding plane of network devices such as switches and routers. The controller uses the OpenFlow protocol to control network devices and choose the best path for application traffic. Because the network control plane can be programmed, contrary to the firmware of hardware devices, network traffic can be managed more dynamically and at a much more granular level.

Centralized control allows the SDN core controller to define the data flows [1]. Each flow through the network must first get permission from the controller, which verifies that the communication is permissible by the network policy [12].

Flow Table. The OpenFlow switch (OF switch) contains the flow tables, which are used to perform packet lookups and forwarding [12]. Using OpenFlow protocol, the controller can add, update, and delete flow entries in the flow table, both reactively (in response to packets) and proactively [12]. Each flow table in the switch contains a set of flow entries. Each flow entry consists of matching fields, counters, and a set of instructions to apply to matching packets [1]. If a packet matches the fields defined in the flow table, the instructions (i.e., "actions") are executed. If no match is found, a packet may be forwarded to the controller or continue to the next flow table.

Packet-In Message. For all packets that do not have a matching flow entry, a packet-in event may be sent to the controller. There are mainly two situations that produce these messages: a mismatch in the tables of the switch or a time to live

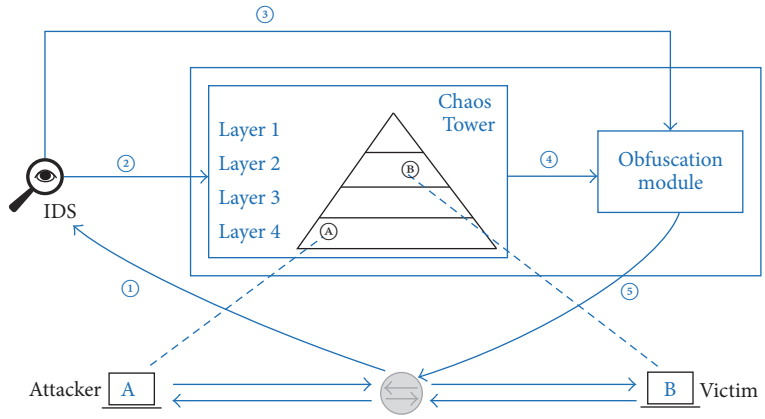


Figure 1: Overall system of CHAOS. ① shows that IDS is monitoring the flows of OF switch; ② and ④ shows that whether normal connections detected by IDS will be obfuscated or not is determined by Chaos Tower; ③ means the abnormal connections detected by IDS will be obfuscated directly; ⑤ means that the obfuscation is executed by OF switches.

(TTL) error [13]. Packet-in messages contain a variety of information about the flow.

After receiving the packet-in message, the core controller decides how to process irregular flows by dispatching a packet-out message.

Packet-Out Message. Packet-out messages are sent from the controller to a switch when the controller wishes to instruct the switch to send packets via a specified port of the switch or to instruct the switch how to forward packets received via packet-in messages.

In CHOAS, we use SDN features and its asynchronous messaging mechanism to implement our dynamic and proactive defense system.

2.2. Threat Model. In most cases, adversaries start an attack on an intranet by collecting as much information about the network as they can. Then they connect to those vulnerable hosts and send attack payloads. Our system, CHAOS, aims to build a dynamic and variable network, so as to defeat reconnaissance attacks on an intranet. Thus, we assume an adversary can scan a network and monitor the network traffic. Moreover, the adversary can eavesdrop network packets. We also assume the protected networks are able to support OpenFlow-based SDN switches and controllers.

3. CHAOS Design

In this section, we provide an overview of CHAOS and then highlight the design of Chaos Tower Structure (CTS).

3.1. CHAOS System Overview. The overall system is illustrated in Figure 1. We design two main modules: Chaos Tower Structure (CTS) and Chaos Tower Obfuscation (CTO) module. CTS defines the communication rules of hosts in a network. The communications that break the CTS rules will be obfuscated using CTO that implements obfuscation mechanisms. We do not obfuscate all network traffic because it will dramatically degrade network performance. In CHOAS, the network traffic will be first sent to IDS, such as Bro. If IDS judges that the traffic is suspicious, CTO module will obfuscate them through installing new flows into OpenFlow switches or modifying flows. Otherwise, if the traffic is judged normal, it will be redirected to our Chaos Tower Structure module. The reasons for doing this are that adversaries often can bypass IDS through some unknown vulnerability attacks. CTS judges the risk of flows and divide them into expected connections and unexpected connections, detailed in Section 3.2.1. Expected connections will be allowed. The unexpected connections will be obfuscated by obfuscation module according to different obfuscation levels.

Chaos Tower Structure (CTS). It is the module we design in the system to determine the communication rules. CTS builds a host hierarchy according to security level of information assets. The tower consists of several layers. Generally, important workgroups are placed in higher layers, whereas unimportant workgroups are placed in lower layers. The importance of every single node which can correspond to a host as well as the host cluster is determined based on the importance degree of services and the vulnerability assessment score in the node. Then we build our model to control network traffic by defining which pairs of hosts can communicate in our topology. Further, according to the tower, the system divides connections into two types: expected and unexpected connections.

Chaos Tower Obfuscation (CTO). It works on the basis of the CTS. It will obfuscate the suspicious connections detected by IDS and unexpected connections detected by CTS. Those connections will be divided into corresponding obfuscation levels. Then CTO obfuscates the connections according to the level.

We next elaborate the major processes of the whole system as shown in Figure 1. If an attacker tries to launch a request from a workgroup in relatively lower layers to a workgroup in higher layers, as indicated by A and B in Figure 1, the system examines the corresponding connection. Firstly, the IDS detects the request and then determines

whether it is a normal connection (Line ①). If it is suspicious, the connection will be directly obfuscated directly (Lines ③ and ⑤). Otherwise, CTS starts to work (Line ②). As shown before, CTS will judge the connection according to its rules. Once the connection is judged to be unexpected by CTS, it will be obfuscated by CTO (Lines ④ and ⑤). In Figure 1, the request is unexpected; as a result, the connection will be obfuscated and B is protected from being scanned or attacked.

3.2. Chaos Tower Structure and Its Workflow. The CTS is a combination of a tree structure and an oriented graph structure. We use a multibranch tree in which to store the workgroup (a host is assigned to a specific workgroup according to its function or importance degree) and the tree defines the privilege of every workgroup. This ensures that most of the layer-jumping behavior is obfuscated. Nonetheless, some layer-jumping behavior is necessary (e.g., the two-way communication between a web server and a database server is necessary, although they are in distinct workgroups). We can define or modify the information conveniently by editing the "Chaos Tower configure file" in the controller to add the special rules. The tower structure with its strict hierarchy enables a more secure and more reliable network.

3.2.1. Tower Construction. In CHAOS, every host or subnet group will be examined and thus a corresponding risk level will be calculated. Risk levels are based on the underlying security metrics. In our system, we use the base score of Common Vulnerability Scoring System (CVSS) [14] to determine the intrinsic qualities of vulnerability. CVSS base score includes two factors, *exploitability of vulnerability* and *impact of vulnerability*. CVSS classifies all the vulnerabilities depending on their features and effects and thus concludes several different kinds of vulnerabilities, such as SQL injection and buffer overflow. For all these kinds of vulnerabilities, CVSS assigns different score to signal the importance of the vulnerability. And in addition to CVSS score, another critical factor is service importance value (SIV). Normally, some hosts are more valuable than others. Thus, we adopt service importance value to represent service's inherent value. It is worth mentioning that, in different networks, the same service may be valued different. That is the reason why we set the SIV table as a part of configuration that administrators should define before the system works. In our system, we introduce the following generic equation to incorporate the CVSS base score and service importance value:

$$\mathrm{RL}(h) = \sum_{v \in V(h)} (\alpha \times \mathrm{SIV}(s) + (1 - \alpha) \times \mathrm{CVSS}(v)), \quad (1)$$

where RL(h) is the risk level of node h; $V(h)$ is a function to return all vulnerability contained in the host h; SIV(s) is a function to return the service importance value of the service s; and CVSS(v) is a function to return the CVSS base score of the vulnerability v. We also introduce the weight coefficient α ($0 \leq \alpha \leq 1$) that allows an administrator to determine how important the service is. The value α can be increased, in which case the service is more important. Otherwise, the administrator can decrease the value of α to weaken the influence of the service but emphasize the influence of the possibility that the hosts would be attacked. According to this given information, we can continue building the original tower, which contains several layers. Each of these layers contains several workgroups, each of which includes several hosts that provide similar functions. CTS also can use some weights such as time, to further define access rules. For example, some access requests can be only allowed in some periods.

After the risk level of each hosts or groups is calculated, we put them into different layers of Chaos Tower. Hosts in the same layers should have the same risk level. Layers with higher risk level will have higher position (e.g., database servers). To deal with the situation that many new devices might well be added to specific subnets, we further divide hosts in the same layer into several groups. Each group contains at least one host. The group division is dependent on the hosts distribution in physical networks. Hence, when there are new devices added to the tower, CHAOS first exams whether they can belong to one existing group or not, if not, its risk level will be calculated and thus it will be mapped onto a new group in the corresponding layers.

In CHAOS, we deem that the more important and risky the host is, the higher the layer it is assigned to. These groups share some common traits; for example, they may be used to store some important network resources. In our system, the administrators can define those important hosts and specify their order of privilege by the risk level of group.

Expected Connections. Expected connections include normal connections and special connections:

(i) Normal connections: they represent the connections from higher layers towards lower layers. In CTS, the communication from higher layers to low layers should be allowed because the hosts of high layer are of high risk level. They often provide important service. So these connections correspond to the allowed communications in an intranet. For those which belong to higher layers only because of their high CVSS score, they can be hardly accessed, which indirectly protect them from being attacked. It is worth mentioning that if the connection from A toward B belongs to normal connections, it does not mean that the connection from B toward A belongs to normal connections.

(ii) Special connections: in order to deal with some special communication request, we define the special connections even though the connections where a host belonging to lower layer accesses a host belonging to higher layers are not judged as normal connections. CTS will judge the special connections as expected connections. We can release special connections temporarily and record them in system log so that administrator can carry out the analysis.

Unexpected Connections. We define unexpected connections as those connections that are not included in the list of expected connections. Generally, these connections are not

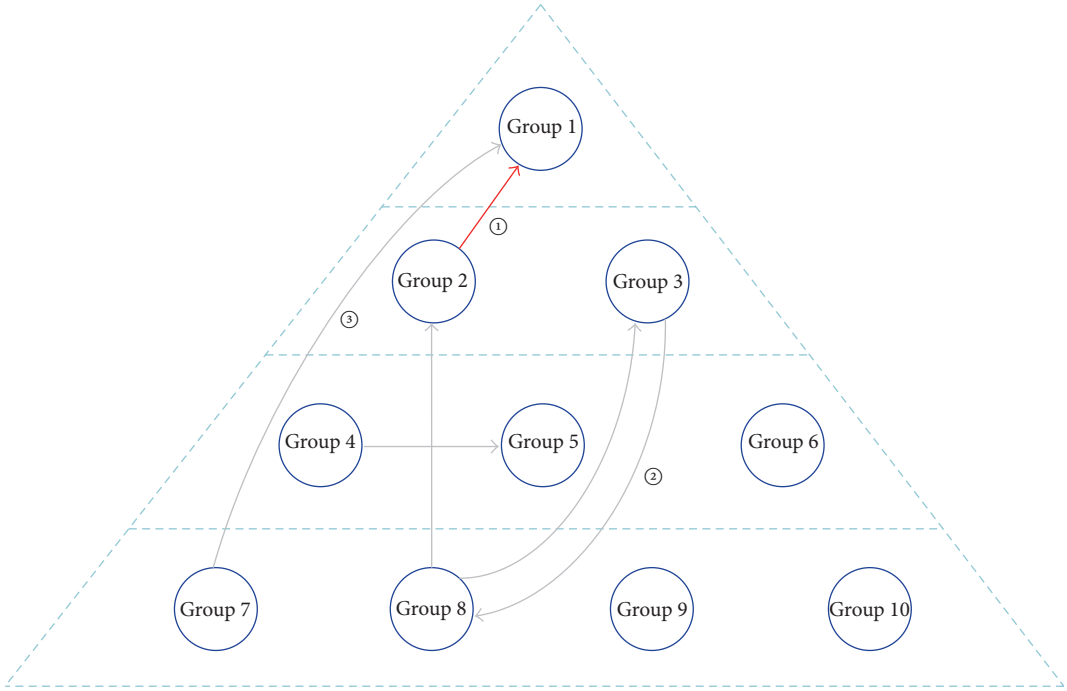


Figure 2: Logical structure of CTS. Red lines like ① represent the unexpected connections; gray lines from upper layers towards lower layers like ② represent the normal connections; gray lines from lower layers towards upper layers like ③ represent special connections.

defined as being allowed and will be detected by our CHAOS system. For example, the connection from a host in employee group toward a host in database group will be judged as unexpected connections.

Here we consider an example to illustrate our proposed CHAOS system in more detail. In Figure 2, Group 1 is placed to the top of tower due to its highest risk level. For line ②, it is a connection from a higher layer to a lower layer, which belongs to normal connections. For line ③, it is a connection from a lower layer to a higher layer but still allowed by CTS, which belongs to special connections. And for line ①, it is an unexpected connection even though it just transgresses only one layer.

3.2.2. Exploiting the Tower. The system reacts differently for expected and unexpected connections.

Expected Connections. We consider expected connections to be legal; thus, the system does not interfere with these connections.

Unexpected Connections. Attention should be paid to these connections. If confronted with an unexpected connection, the controller will send a request to obfuscation module to obfuscate it. Generally, if the connection is established by layer-jumping or occurs within the same layer, it is considered abnormal and will be obfuscated. However, some special connections can be defined by system administrator; these connections cannot be judged as abnormal communication and not be obfuscated.

4. Obfuscation

In our system, we implement three kinds of obfuscations, which are host mutation obfuscation, port obfuscation, and obfuscation based on decoy servers. For unexpected connections judged by CTS and the abnormal connections judged by IDS, our system will grade them and apply corresponding obfuscations according to their degree of abnormality.

Host Mutation Obfuscation. This technique is aimed to defend MITM (Man in the Middle) attack and third-party traffic monitoring by replacing source IP address and destination IP address of the packet to virtual IP addresses when transferring it between switches [15]. The mechanism is shown in the right-hand side of Figure 3. The OpenFlow controller frequently assigns a random virtual IP (vIP) to each real IP (vIP). When *Host1* initiates the connection to *Host2* and sends an initial packet using real source IP (*r1*) and real destination IP (*r2*), the first OF switch that captures the initial packet *(OF switch 1)* encapsulates and sends the packet to SDN controller, where a rIP-vIP mapping table is stored, and maps *r1* and *r2* to corresponding virtual IPs (*v1* and *v2*). When the initial packet reaches the OF switch that is nearest to *Host2 (OF switch n)*, a similar reverse mapping is executed, changing vIPs back to rIPs, namely, *v1* to *r1* and *v2* to *r2*. In this sense, packets in the middle (between *OF Switch 1* and *OF Switch n*) only contain virtual IPs so that real host IPs are concealed.

Port Obfuscation. This technique is aimed to defend port-scanning-based attack. In this case we inject some entirely fake information into responses as well as hiding some real

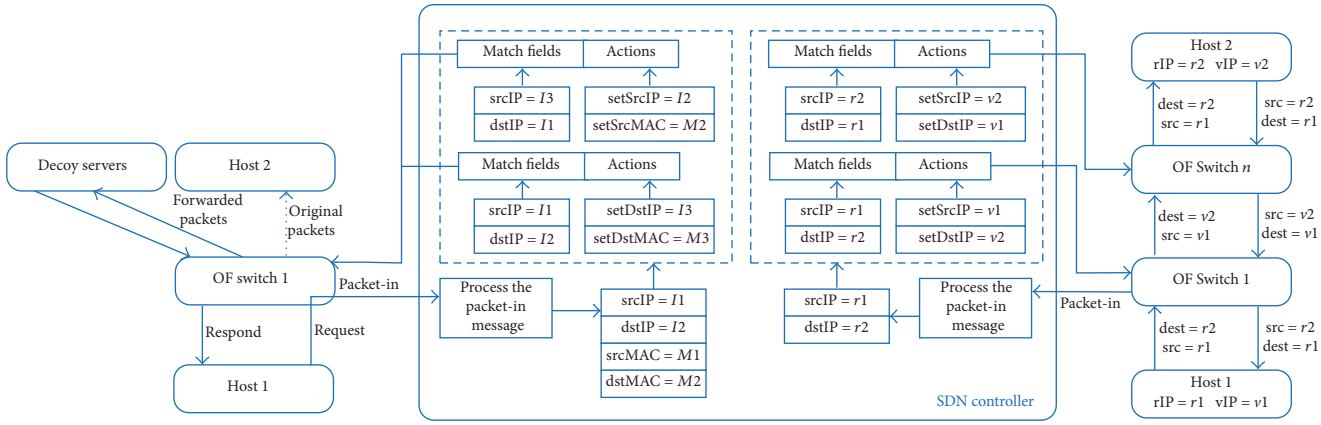


Figure 3: Mechanism of host mutation and decoy-servers-based obfuscation.

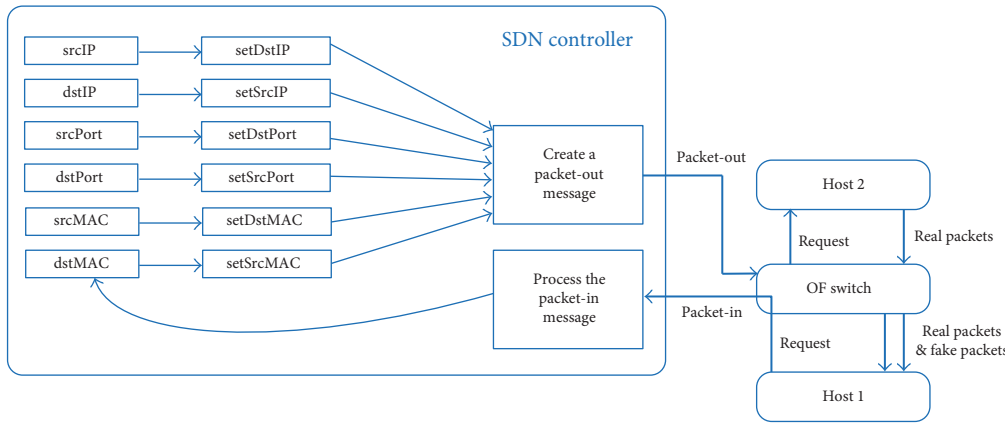


Figure 4: Mechanism of port obfuscation.

information. As is shown in Figure 4, when IDS detects a port scanning, CHAOS system will inject fake packets into the real packets by generating corresponding acknowledgment to obfuscate the result of the port scanning. For instance, when a TCP scan is detected and port obfuscation is applied, the TCP packets will be fetched by switch and sent to the controller through packet-in. Then the controller will analyze the packet, generate a corresponding packet-out, and send it to the switch. The acknowledgments of some injected packets are 0, while some are 1. Whether to inject or modify the packets is generally on a random basis. Therefore, the results of port scanning will show a certain degree of randomness and fuzziness.

Obfuscation Based on Decoy Servers. In CHAOS system, we deploy a number of decoy servers as an attack trap. In most cases, decoy servers can even delay the attack. When applying this strategy, our system will forward the unexpected connections to the decoy servers. As is shown in Figure 3, when a host launches a request, our system can analyze the packets and install flows into the switch, which will forward the unexpected connections to our decoy servers. In this way, suspicious users can only access various decoy servers. The services we deployed in the decoy servers can further help us discover the real attackers.

These three obfuscation strategies are applied under different circumstances. In the tower, we use the threshold factor to determine which strategy is applied. It is determined by calculating the ratio of leapfrog access number to the total number of the layers, named altitude. If the altitude of the connection is smaller than threshold, the connection will be obfuscated later. If not, the connection will be forwarded to decoy servers. In short, the threshold factor divides unexpected connections into two parts by altitude. Connections which belong to the first part will be obfuscated, while connections which belong to the other part will be forwarded to decoy servers. Administrators can change the threshold factor depending on the security level and structure of the network. The threshold factor assures that attacks will be obfuscated in theory.

In addition, we introduce a parameter named RandomIndex ($0 \leq \text{RandomIndex} \leq 1$) to define the possibility of CHAOS performing obfuscation; that is, the closer the RandomIndex to 0, the higher the likelihood of CHAOS injecting fake information into the network. We define srcLayer as the layer in which the host launches the request and dstLayer as

```
Require: packetInp, Inf, Sup, RandomIndex; {HEIGHT is the height of the tower}
  if isFromSrcSwitch (p) orisFromDstSwitch (p) then installHostMutationFlows (p);
  end if
  srcLayer ← getSrcLayer (p);
  dstLayer ← getDstLayer (p);
  Δ Altitude ← srcLayer – dstLayer;
  Possibility ← random [0, 1];
  if Δ Altitude ≥ 0 then
    Forward (p);
  else
    Δ Altitude ← −Δ Altitude;
    if Δ Altitude/HEIGHT ≤ threshold then
      if isRequestPacket (p) andPossibility ≥ RandomIndex
      then
        PacketOut (p);
      else
        ForwardToDecoyServer (p);
      end if
    else
    InstallForwardingFlows (p);
    end if
  end if
```

Algorithm 1: CHAOS.

the layer in which the host responds. Then we define altitude as the difference in height between these two respective layers (i.e., the height of srcLayer minus the height of dstLayer). RandomIndex assures that obfuscation is random so that attackers will not notice our system immediately.

Our design of obfuscation contains two aspects. First, as most network mapping tools perform their operations by using ICMP packets and TCP or UDP scans, ICMP messages are typically used to verify connectivity or reachability of potential targets. TCP and UDP port scans are used to identify running services of a target. Replies (TCP RST, silent drop, or ICMP unreachable) to scans can also reveal what services are allowed or filtered through transit devices. Additionally, the TTL field of IP packets is used to identify the hop distance between the target and the destination. SDN-enabled devices can be used to confuse the reconnaissance. For example, traffic to a destination that can be blocked according to a filtering policy can be silently dropped and SDN utilities can generate varying responses that will confuse the attacker. In the case of traffic that is permitted by the filtering policy (that is, it is legitimate), the SDN policy does not interfere. The action for each packet is kept in a buffer to ensure consistent behavior. As a result of this algorithm, random ports will appear to the scanner as being open. Digging deeper in order to identify services running on these fake open ports would require more resources from the attacker [16]. Secondly, the controller determines the type of connection (i.e., via srcIP or dstIP) and installs necessary flows in all OF switches in the path. These flows will change the srcIP and dstIP of each packet (assuming srcIP changed to be vsrcIP and dstIP changed to be vdstIP) so that the packet will be different from what they actually are. But meanwhile, these flows will also make sure that the packet can be sent to the destination host by changing the vsrcIP and vdstIP to srcIP and dstIP in the end. Each connection must be associated with a unique flow, because the rIP-vIP translation changes for each connection. This property guarantees the end-to-end reachability of hosts, because the rIP-vIP translation for a specific connection remains unchanged regardless of subsequent mutations [15].

The process is presented as Algorithm 1. Here we use a pseudo-code to clarify the process. Firstly, if we find that the packet-in message comes from the source switch or destination switch of the packet, we will install flow tables of host mutation. Then, the connection will be judged to be obfuscated or not. For expected connections, the packet will be forwarded directly. But for unexpected connections, the packet will be obfuscate or forwarded to a decoy server if the altitude is bigger than the threshold configured by administrator.

5. Implementation and Evaluation

5.1. System Implementation. The structure of our system is shown in Figure 5. The routing was managed entirely by the Floodlight controller and monitored by Bro. We implemented three modules. The first one we implemented is the Chaos Tower module, the purpose of which is to build the Chaos Tower and get unexpected flows. Then, we implemented the obfuscation module in Floodlight, which obfuscates the unexpected flows and abnormal traffic judged by IDS. Finally, we implemented the CHAOS management module which allows administrators to further configure their networks.

We provide an implementation of obfuscation with Bro's warning message. In the beginning, we push flow tables into switches so that all flows are allowed. Then, we use Bro to monitor the network. When suspicious flows are detected,

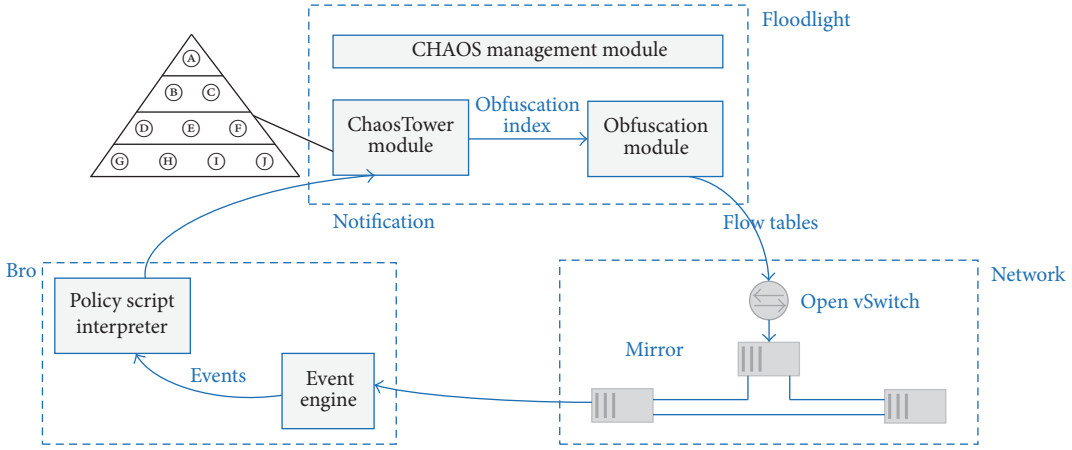


Figure 5: System implementation.

the tower will determine the corresponding obfuscation index and transfer it to obfuscation module. After that, corresponding flow tables will be updated to make sure that the obfuscation works in the network.

5.2. Scanning and Foot-Printing Test. Foot-printing and scanning are techniques for gathering information about computer systems in networks. These techniques are implemented by various security auditing tools as the first step when launching an attack. Nmap [17] and the scanner modules in Metasploit [18] contain many payloads to gather sensitive information from target machines, whereas Nessus [19] and WVS (Web Vulnerability Scanner) focus on vulnerability detection and exploitation.

In our test, we used Nmap to evaluate the information obfuscation ability of CHAOS. Nmap uses raw IP packets in novel ways to determine which hosts are available on the network, which services (application name and version) those hosts are offering and which operating systems (and OS versions) they are running, which type of packet filters/firewalls are in use, and many other characteristics [17]. Our test involved configuring some vulnerable hosts in the network, after which we used Nessus to detect vulnerabilities to test whether CHAOS would be able to confuse and deceit Nessus.

We tested the performance of our system by launching a series of attacks under different circumstances. We consider three situations against Nmap. In the first, the network was unprotected; in the second, we implement a fully obfuscated system [16]; and in the third, our CHAOS system was implemented. When simulating the attack, we used Nmap to scan the entire network several times. Based on its response and the reality of its given circumstances, we concluded the result (Figures 6 and 7). Besides this, we used a ping command to test the effect of our system on normal traffic (Figure 8).

5.3. Results. We carried out our experiments in CloudLab [20] and deployed the network shown in Figure 2.

First, we used Nmap to determine whether our CHAOS system was able to deceit the security tool. There are two situations involved in this experiment. We selected the hosts of Group 4 and Group 3 in Figure 2; thus, the obfuscation index is 0.5, so obfuscation based on decoy servers will work then.

We define information disclosure percentage (IDP) as our index and calculate it by the following formulas. ID is the amount of information that the adversary fetches from the victim. NONE represents the unprotected network. FON represents the fully obfuscated network. CHAOS represents the network protected by CHAOS.

$$\begin{aligned} \mathrm{IDP}_{\mathrm{CHAOS}} &= \frac{\mathrm{ID}_{\mathrm{CHAOS}}}{\mathrm{ID}_{\mathrm{NONE}}}, \\ \mathrm{IDP}_{\mathrm{FON}} &= \frac{\mathrm{ID}_{\mathrm{FON}}}{\mathrm{ID}_{\mathrm{NONE}}}. \end{aligned} \tag{2}$$

Figure 6 shows the percentage of information disclosure of an unprotected network and a network (Level 2) protected by CHAOS as a function of the number of times the network was scanned by Nmap. The figure shows that, for the network protected by CHAOS, the percentage of information disclosure is decreased effectively.

Secondly, we studied the correlation between the degree of threat of the adversary and the information disclosure he would experience. For comparison, we implemented another MTD system, fully obfuscated network, which obfuscates all the packets in the network. Figure 7 shows the information disclosure in an unprotected network, a network protected by CHAOS, and fully obfuscated network [15], all of which face different degrees of threats. The fully obfuscated network obfuscates all the packets by some static policies. Thus, it is able to decrease information disclosure when the threat reaches a certain degree, but does not decrease information disclosure further when the degree of threat is elevated beyond that certain degree, because of its static solution. However, the network protected by CHAOS decreases information disclosure when the degree of threat is elevated. Only

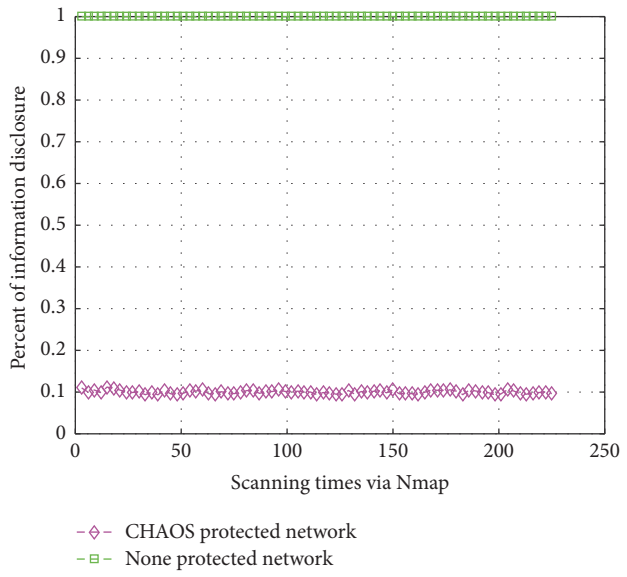


Figure 6: Information disclosure with scanning time.

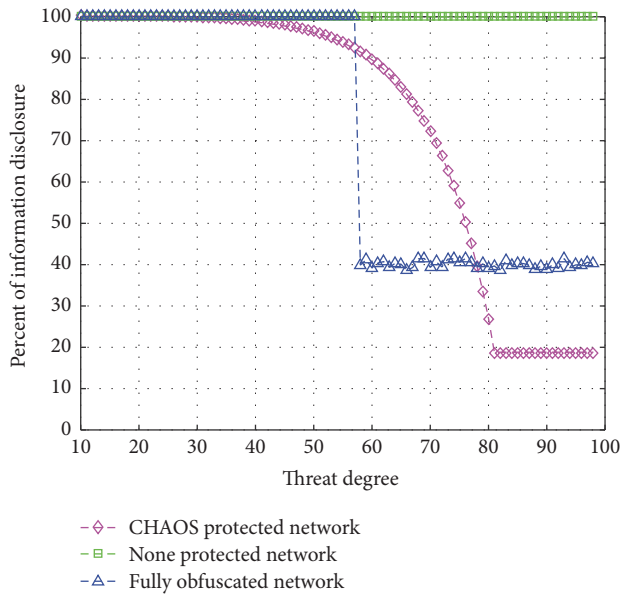


Figure 7: Information disclosure with respect to threat degree.

a few information disclosures exist when the threat reached a very high degree.

After that, we compared the performance cost of the three networks. As above, we compare the network protected by CHAOS with the unprotected and fully obfuscated network. We use the example shown above to test the performance of these systems and to measure the average delay time of the connections under each system. Figure 8 shows the delay time of the unprotected network, the network protected by CHAOS, and fully obfuscated network with changing package counts. We conclude that both the networks protected by CHAOS and fully obfuscated network increase the delay time to some extent, although the network protected

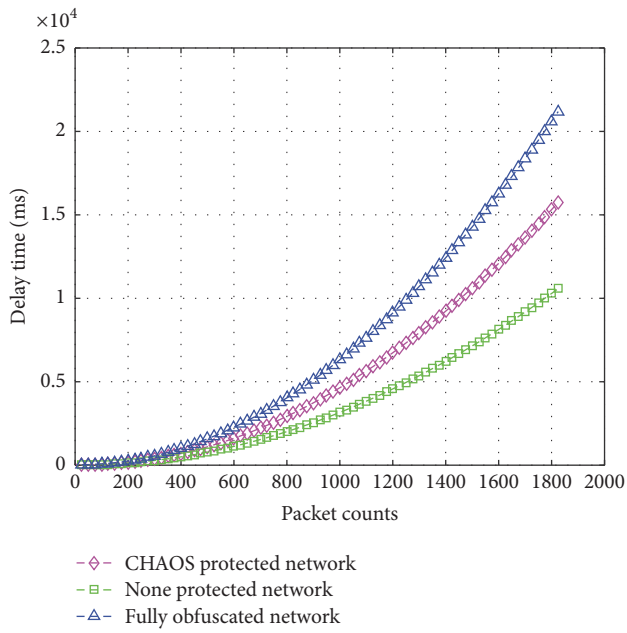


Figure 8: Delay time with respect to packet count.

by CHAOS has a reduced delay compared to that fully obfuscated network. Thus, our system enables the network to perform faster. We discovered that the transforming speed of our system is faster than that of random obfuscation system especially when the network is crowded.

The result above can be understood in terms of the following factors.

First, we use Bro to monitor the network and transfer those suspicious flows. The important point is that Bro runs stand-alone so it makes quite few effects to the speed of the network.

Then, the Chaos Tower is also a factor that reduces the delay time. We assume that the Chaos Tower is to be built as a binary tree in the network and the number of layers is L; hence,

$$N = 2^L - 1. \tag{3}$$

We consider a situation in which each workgroup sends a request to the remaining groups, which means that the sum of the connections the unprotected situation and the MTD solution would have to process would be

$$\begin{aligned} C_{\text{NONE}} &= 0, \\ C_{\text{MTD}} &= N * (N - 1). \end{aligned} \tag{4}$$

However, we only need to obfuscate the connections from the lower layers toward the higher layers in our CHAOS system, the number of which is

$$C_{\text{CHAOS}} = \sum_{i=1}^{L-1} \left(2^i * \left(2^i - 1\right)\right). \tag{5}$$

In the end, we launched several real attacks to testify robustness of our system. We employ some vulnerable hosts in the network. In the experiment, MS 08-067

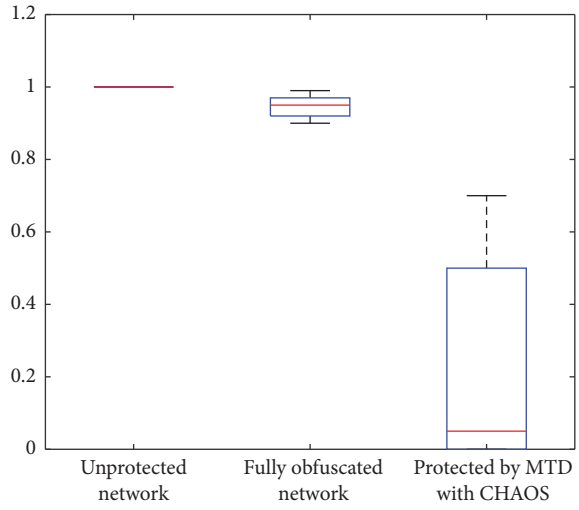


Figure 9: Attack testing.

is the vulnerability that we test. The hosts can be easily attacked by any pen-testing tools which contain payload of MS 08-067. Actually, in Chaos Tower, we employ a vulnerable host in each layer. Then we use one of them to play the role of attacker in turn. Figure 9 shows the results of the unprotected network, the network protected by CHAOS, and the fully obfuscated network. We conclude that, in the network protected by CHAOS, only a few attacks directed to hosts belonging to adjacent layers succeeded. However, in the fully obfuscated network, most attacks succeeded in the end. The worst is that almost all attacks succeeded in the unprotected network. Thus, our system can decrease the success rate of such kind of attacks significantly.

6. Related Work

Several researchers have reported work on MTD. Kewley et al. [21] performed the initial research in the area of dynamic network defense and proved that dynamic network reconfiguration, such as randomly changing the IP address and port numbers, would effectively inhibit an adversary's ability to gather intelligence and thus degrade the ability to successfully launch an attack. Al-Shaer proposed MUTE, a moving target defense architecture [5], which implements the moving target through random address hopping and random finger printing. Furthermore, they presented BDD, a model for creating a valid mutation of network configuration. Zhuang et al. [4] investigated the application of moving target defenses to network security and presented a high-level architecture of the MTD system. Their simulation results show the potential for MTD to be effective in preventing attacks against computer networks. Furthermore, they proposed a formal theory to describe the MTD system and its basic properties and formalized the MTD entropy hypothesis, which states that the greater the entropy of the system configuration, the more effective the MTD system [22, 23]. Stallings proposed the use of SDN in the implementation of MTD mitigations. Al-Shaer et al. [15] proposed OpenFlow Random Host Mutation (OF-RHM), which uses OpenFlow to develop an MTD architecture that transparently mutates host IP addresses with high unpredictability, while maintaining configuration integrity and minimizing operational overhead.

However, current network-based MTD obfuscates networks indiscriminately that makes some networks services unavailable, for example, some key services like web and DNS, because some information of these services has to be opened to the outside and remain real. If MTD obfuscates these services fully, it will return users with virtual IPs and ports, making these services unable to use. Moreover, obfuscation will affect the performance of networks. To obfuscate hosts indiscriminately will severely reduce the performance of networks undoubtedly. In contrast to the above work, CHAOS discriminately obfuscates hosts with different security levels in networks.

Zhang et al. [24] proposed to construct an incentive compatible moving target defense by periodically migrating virtual machines (VMs), thereby making it much harder for adversaries to locate the target VMs. Gillani et al. [25] proposed to defend against DDoS attacks by migrating virtual networks (VNs) to dynamically reallocate network resources. Different from their work, CHOAS leverages SDN features to obfuscate network information instead of migrating target objects.

Previous research involving memory address space randomization [26–28], instruction set randomization [29], and software diversification [30, 31] also used the idea of a moving target to increase the attack difficulty and cost by enlarging the exploration surface or moving the attack surface. The objective of our work is to enhance network security; hence, the aspects mentioned here are not discussed in detail.

7. Conclusion

MTD is able to create a type of changing network so as to increase the difficulty and cost for an adversary aiming to launch a network attack. In this paper, we proposes an SDN-based MTD system named CHAOS which discriminately obfuscates hosts with different security levels in networks so as to keep some key services available and low performance cost. CHAOS incorporates the Chaos Tower Structure to represent a hierarchy of all the hosts on the network and leverages SDN features to obfuscate the attack surface to enhance the unpredictability of the networking environment. CHAOS offers rapid obfuscation of unexpected network traffic but does not interfere with normal traffic. The evaluation shows that a network protected by CHAOS can effectively lower the percentage of information that is disclosed.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (nos. 61272452, 61332019, and 61402342), the National High-Tech Research and Development Program

of China ("863" Program) (no. 2015AA016002), and the National Basic Research Program of China ("973" Program) (no. 2014CB340601).

References

[1] Open Networking Foundation, "OpenFlow1.1.0 specification," 2011, http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[2] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Adversary-aware IP address randomization for proactive agility against sophisticated attackers," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '15)*, pp. 738–746, IEEE, April 2015.

[3] Y. Wang, J. Bi, and K. Zhang, "A tool for tracing network data plane via SDN/OpenFlow," *Science China Information Sciences*, vol. 60, no. 2, Article ID 022304, 2017.

[4] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *Proceedings of the 2013 6th International Symposium on Resilient Control Systems, ISRCS 2013*, pp. 162–169, San Francisco, Calif, USA, August 2013.

[5] E. Al-Shaer, "Toward network configuration randomization for moving target defense," in *Moving Target Defense*, vol. 54 of *Advances in Information Security*, pp. 153–159, Springer, New York, NY, USA, 2011.

[6] J. Sun and K. Sun, "DESIR: Decoy-enhanced seamless IP randomization," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications, IEEE INFOCOM 2016*, April 2016.

[7] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, Calif, USA, February 2016.

[8] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan, "PSI: Precise Security Instrumentation for Enterprise Networks," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*, San Diego, Calif, USA, February 2017.

[9] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[10] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Enabling Practical Software-defined Networking Security Applications with OFX," in *Proceedings of the Network and Distributed System Security Symposium (NDSS '16)*, San Diego, Calif, USA, February 2016.

[11] Stanford University, "Clean slate program," http://cleanslate.stanford.edu/.

[12] Open Networking Foundation, "OpenFlow switch specification," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[13] Flowgrammable Team, "Packet in messages," 2014, http://flowgrammable.org/sdn/openow/message-layer/packetin.

[14] P. Mell, K. Scarfone, and S. Romanosky, "A Complete Guide to the Common Vulnerability Scoring System Version 2.0," 2007, https://www.nist.gov/publications/complete-guide-common-vulnerability-scoring-system-version-20.

[15] E. Al-Shaer, Q. Duan, and J. H. Jafarian, "Random host mutation for moving target defense," in *Security and Privacy in Communication Networks*, A. D. Keromytis and R. Di Pietro, Eds., vol. 106 of *Lecture Notes of the Institute for Computer Sciences*, pp. 310–327, Springer, Berlin, Germany, 2013.

[16] P. Kampanakis, H. Perros, and T. Beyene, "SDN-based solutions for Moving Target Defense network protection," in *Proceedings of the 15th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM '14)*, pp. 1–6, Sydney, Australia, June 2014.

[17] G. Lyon, Network mapper, 2017, https://nmap.org/.

[18] Rapid7 LLC, Metasploit, 2009, https://www.offensive-security.com/metasploit-unleashed/vulnerabilityscanning.

[19] Tenable, Nessus, 2017, http://www.tenable.com.

[20] The CloudLab Team, CloudLab, 2014, http://www.cloudlab.us.project.

[21] D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering," in *Proceedings of the DARPA Information Survivability Conference and Exposition II, DISCEX 2001*, pp. 176–185, Anaheim, Calif, USA, June 2001.

[22] R. Zhuang, S. A. DeLoach, and X. Ou, "A model for analyzing the effect of moving target defenses on enterprise networks," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference (CISRC '14)*, pp. 73–76, April 2014.

[23] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense (MTD '14)*, pp. 31–40, ACM, November 2014.

[24] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang, "Incentive compatible moving target defense against VM-colocation attacks in clouds," in *Information Security and Privacy Research*, pp. 388–399, Springer, Berlin, Germany, 2012.

[25] F. Gillani, E. Al-Shaer, S. Lo, Q. Duan, M. Ammar, and E. Zegura, "Agile virtualized infrastructure to proactively defend against cyber attacks," in *Proceedings of the 34th IEEE Annual Conference on Computer Communications and Networks, IEEE INFOCOM 2015*, pp. 729–737, May 2015.

[26] F. P. Miller, A. F. Vandome, and J. McBrewster, *Address space layout randomization*, Alphascript Publishing, 2010.

[27] K. Chongkyung, J. Jinsuk, C. Bookholt, X. Jun, and N. Peng, "Address Space Layout Permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC 2006*, pp. 339–348, December 2006.

[28] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security (CCS '04)*, p. 298, Washington, DC, USA, October 2004.

[29] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis, "On the general applicability of instruction-set randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 3, pp. 255–270, 2010.

[30] Y. Huang and A. K. Ghosh, "Introducing diversity and uncertainty to create moving attack surfaces for web services," in *Moving Target Defense*, vol. 54 of *Advances in Information Security*, pp. 131–151, Springer, New York, NY, USA, 2011.

[31] M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, "End-to-end software diversification of internet services," in *Moving Target Defense*, vol. 54 of *Advances in Information Security*, pp. 117–130, Springer, New York, NY, USA, 2011.