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

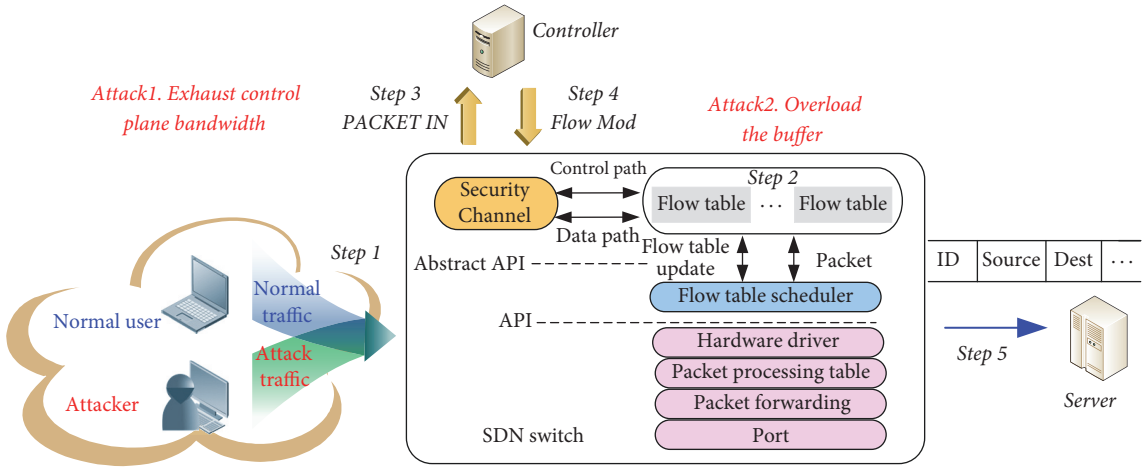

Figure 1: SDN architecture and denial-of-service attacks.

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

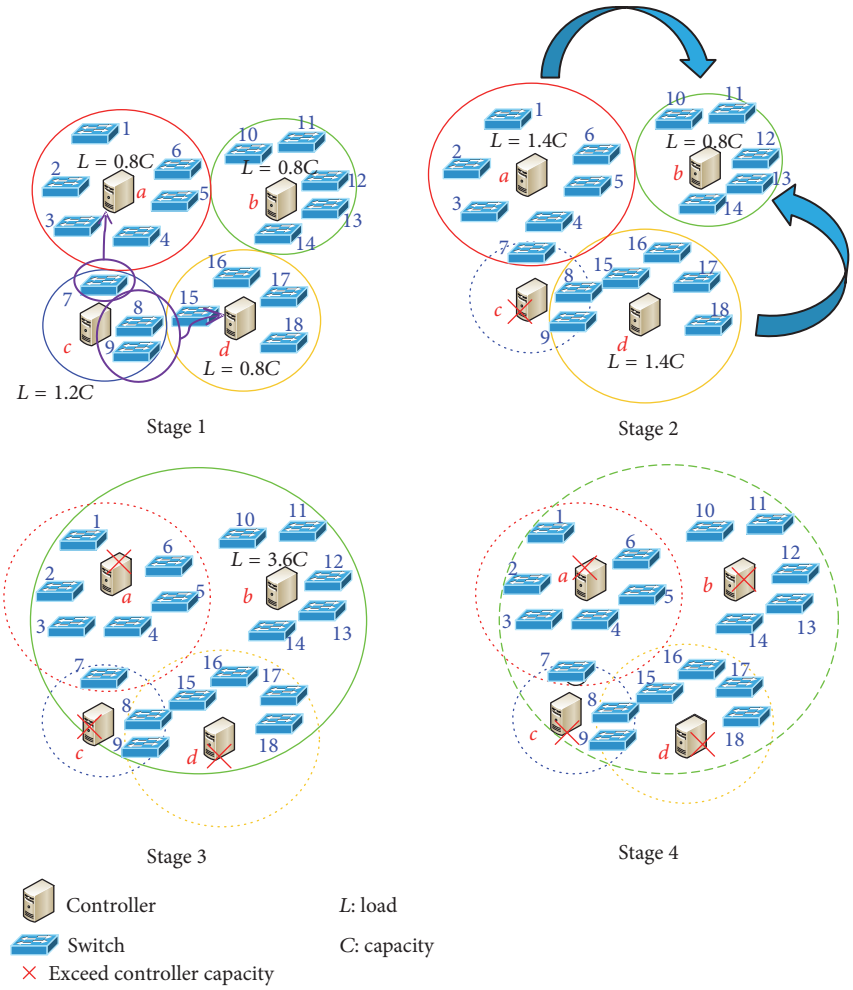

Figure 2: The cascading failures of controllers.

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

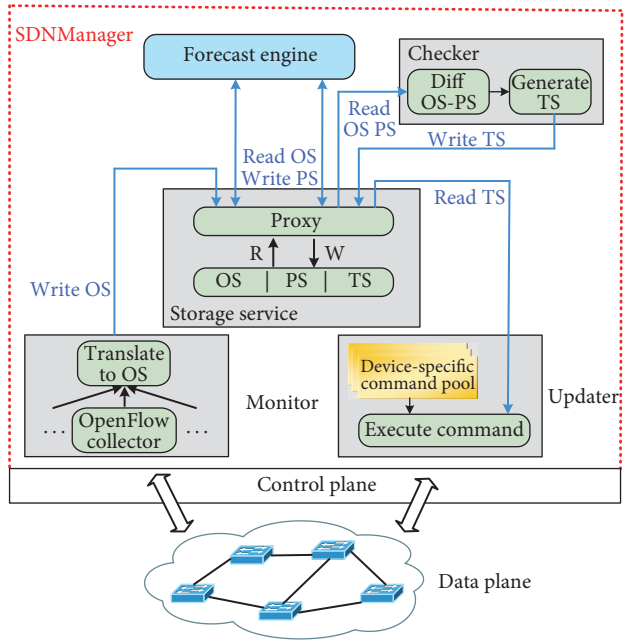

Figure 3: The architecture of SDNManager.

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

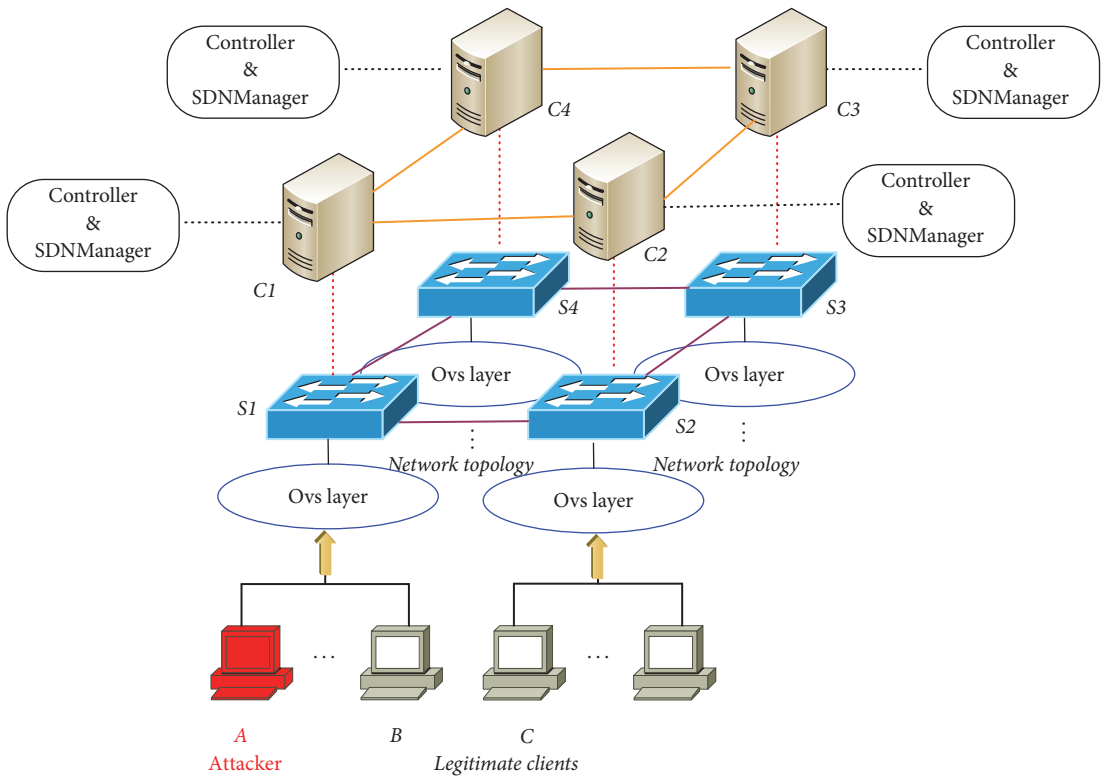

Figure 4: The topology used in the experiments.

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).

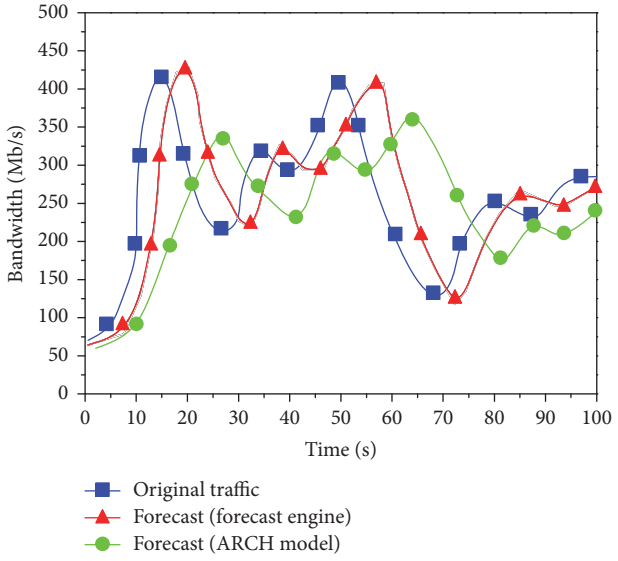

Figure 5: DTS and ARCH forecast of sample trace.

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.

$$\text{ForecastError}\,(\%) = \frac{100}{n}\sum_{t=1}^{n}\left|\frac{z_t - \widetilde{z}_t}{z_t}\right|. \tag{21}$$

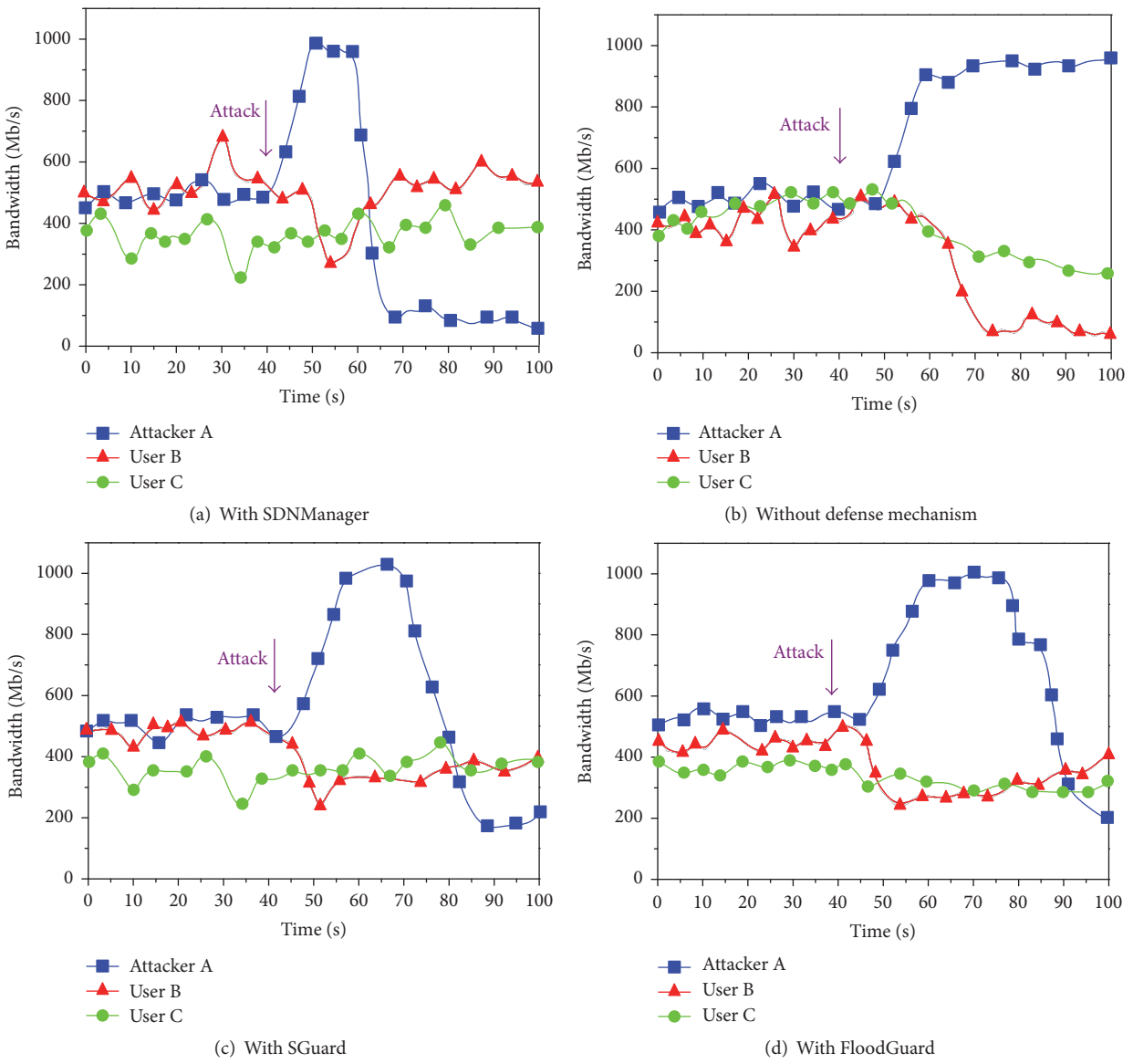

FIGURE 6: The bandwidth changes before and after the attack.

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

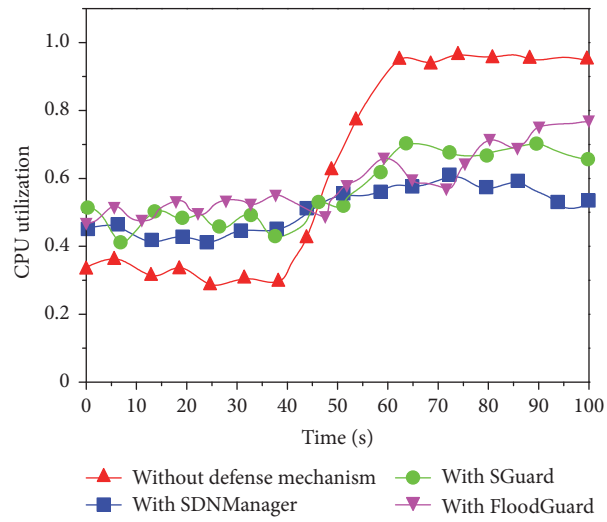

Figure 7: CPU utilization.

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

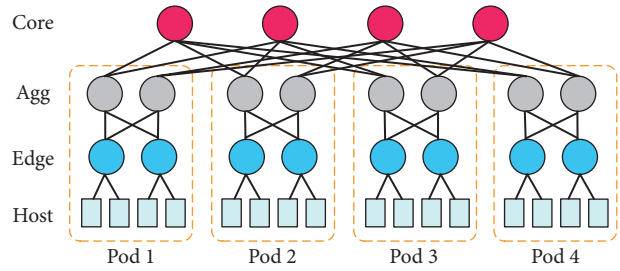

Figure 8: 4-pod fat-tree (example topology).

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

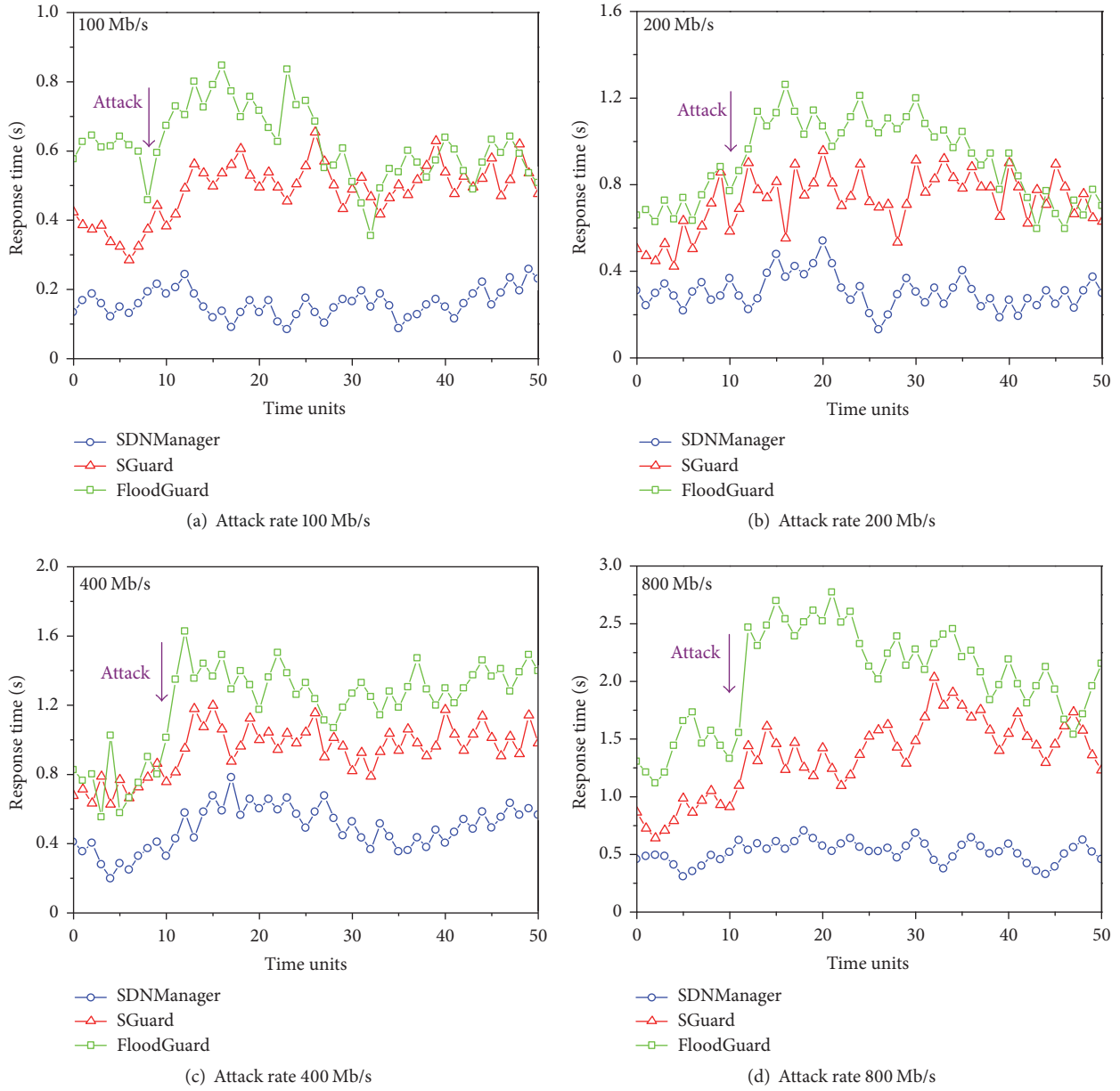

Figure 9: Response time comparison using dynamic controller scheduling strategy.

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

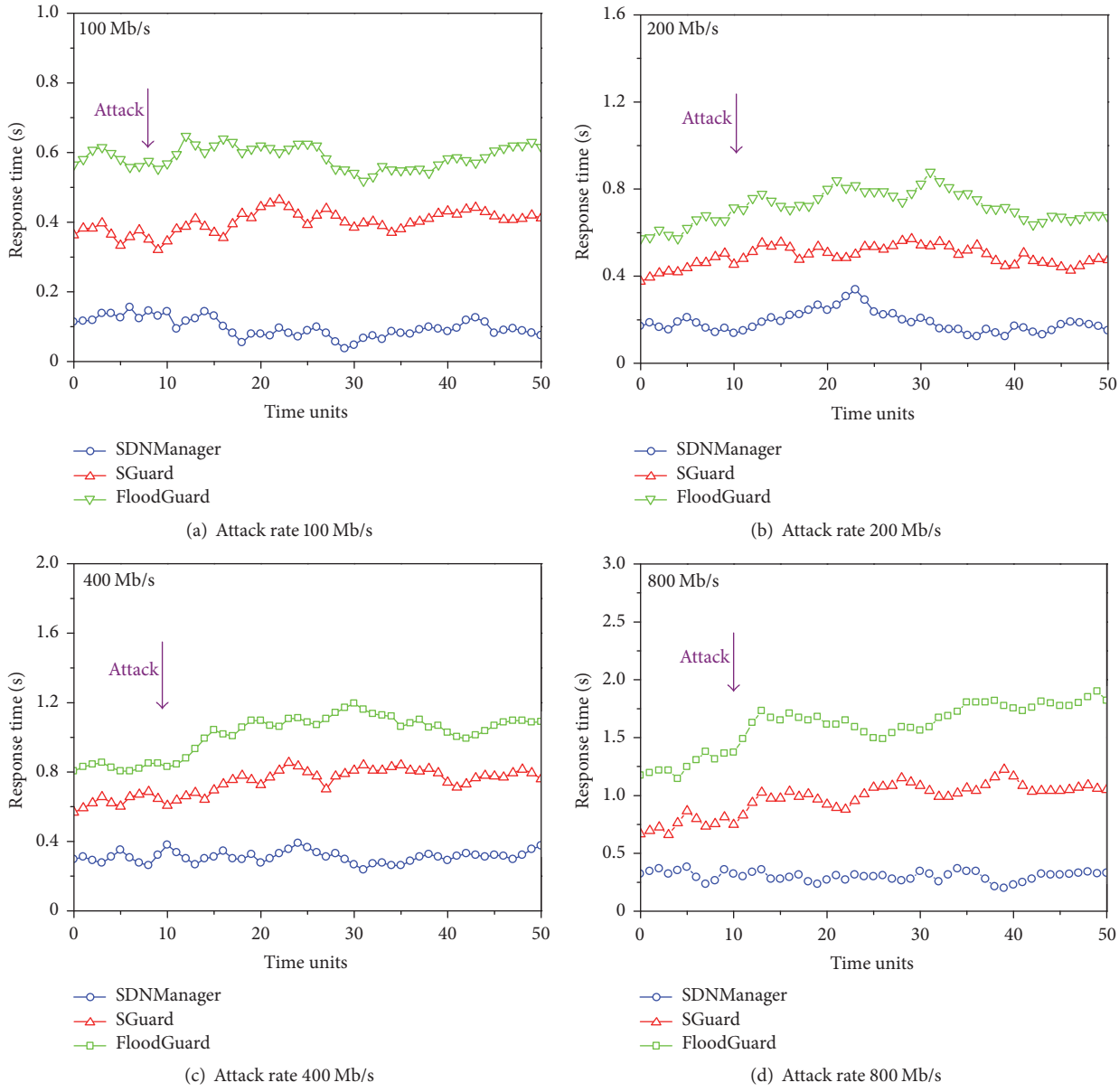

Figure 10: Response time comparison using dynamic controller scheduling strategy.

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.