Information Sciences 327 (2016) 288–299

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Leveraging software-defined networking for security policy enforcement

Jiaqiang Liu[a], Yong Li[a,*], Huandong Wang[a], Depeng Jin[a], Li Su[a], Lieguang Zeng[a], Thanos Vasilakos[b]

[a] Department of Electronic Engineering, Tsinghua University, Beijing, China
[b] Department of Computer Science, Kuwait University, Safat, Kuwait

ARTICLE INFO

Article history:
Received 9 June 2014
Revised 21 July 2015
Accepted 15 August 2015
Available online 21 August 2015

Keywords:
Software defined network
Security
Network update

ABSTRACT

Network operators employ a variety of security policies for protecting the data and services. However, deploying these policies in traditional network is complicated and security vulnerable due to the distributed network control and lack of standard control protocol. Software-defined network provides an ideal paradigm to address these challenges by separating control plane and data plane, and exploiting the logically centralized control. In this paper, we focus on taking the advantage of software-defined networking for security policies enforcement. We propose a two layer OpenFlow switch topology designed to implement security policies, which considers the limitation of flow table size in a single switch, the complexity of configuring security policies to these switches, and load balance among these switches. Furthermore, we introduce a safe way to update the configuration of these switches one by one for better load balance when traffic distribution changes. Specifically, we model the update process as a path in a graph, in which each node represents a security policy satisfied configuration, and each edge represents a single step of safely update. Based on this model, we design a heuristic algorithm to find an optimal update path in real time. Simulations of the update scheme show that our proposed algorithm is effective and robust under an extensive range of conditions.

1. Introduction

With the rapid development of information technology, we have now entered a networked world, in which we access information, communicate with others and run business, do all of them on the Internet. Security then becomes an important concern for these network systems, e.g., how to prevent malicious attacks and guarantee that the data are only accessible by authorized users. Security is especially important for enterprise and data center [35–37], because they usually host a variety of services and reposit a large volume of sensitive data, e.g., user profiles, company confidential data, which could belong to multiple tenants in cloud environment [6]. If not well protected, the malicious users may get access to these sensitive data or services by exploiting system vulnerabilities like weakness of access control and using some fingerprinting techniques like port scanning, IP spoofing, etc. [42]. To achieve the protection, the operators need to draw up and deploy fine-grained policies like access control, rate-limiting, and communication isolation in the network [39,40].

[*] Corresponding author: Y. Li
E-mail address: liyong07@tsinghua.edu.cn

http://dx.doi.org/10.1016/j.ins.2015.08.019
0020-0255/© 2015 Elsevier Inc. All rights reserved.

Table 1
The example of security policies (IDS: intrusion detection system).

The target traffic	The action
srcIp = 123.4.1.*, dstPort = 22 or 443	Drop
srcIp = 115.6.2.*	Pass
dstIp = 177.8.9.6	Forward to IDS

In traditional networks, middleboxes such as firewalls and Deep Packet Inspection (DPI) are exploited to carry out security policies [2]. There are two approaches to deploy them, i.e., placing them on network paths between end-points [1] or placing them off network paths by connecting them to middle switches [9]. However, both of them are inflexible. In the first approach, inserting new middleboxes is hard due to the restriction of network topology [32]; while in the second approach, installing and updating network configuration are complicated and insecure since the operators cannot directly control packet forwarding [33]. Besides, managing and maintaining these middleboxes are expensive. For example, even a small network with tens of middleboxes requires a management team of 6–25 personnel [43].

The emergence of software-defined network (SDN) offers a great opportunity to address the above issues [22,23]. In the SDN, data plane and control plane are isolated. The operator can program the control plane to directly control packet forwarding in the data plane through standard interfaces, e.g., OpenFlow [14]. This ability enables operators to adopt applications in the control plane to flexibly and automatically configure switches to forward packets to different middleboxes, which significantly simplifies the management and minimizes the misconfiguration [41]. Besides, since OpenFlow supports more actions than forward, such as drop, modify packets header and queue [15], the operators can directly use OpenFlow switches to implement security policies, e.g., installing flow entries on them to directly drop invalid packets defined by the security policies. Furthermore, the controller has a global view of the network [7,11], therefore it is easier to guarantee the consistency and completeness of security policy enforcement. In addition, SDN-based approach reduces the requirement of human based maintenance and hence also saves the operational cost.

Given above benefits, many companies, such as Google [8,16] and VMware [30], have used SDN in their data centers. It is also broadly recognized that SDN will be widely deployed in data centers in the near future [12,17]. Therefore, in this paper, we focus on SDN based security policy enforcement. We assume that the underlying switches are OpenFlow enabled, and there is a logically centralized controller, since the control plane in SDN is expected to be logically centralized. In implementation, the controller can use multiple physical servers [11] for scalability and use FlowVisor [27] for network virtualization. Applications running on the controller generate and install flow entries on OpenFlow switches to customize packet processing.

We first show that most security policies can be transferred into flow entries and deployed on OpenFlow switches. Particularly, since the total number of required flow entries may be millions [39,40] and the flow table size of a single switch is usually limited [38], flow entries need to be deployed on multiple switches. We propose a scalable and efficient architecture with two layer OpenFlow switches for security policy enforcement. In the first layer, packets are classified into different classes and forwarded to different switches in the second layer, according to security policy constraints and load balance considerations. In the second layer, the concrete security policies of different classes are deployed on specific switches. The operator can easily scale the system by adding more switches, and efficiently use the provided processing capability by adjusting the switches assigned to each traffic class.

After that, we show that update the proposed two layer OpenFlow switches may generate security holes. But fortunately, with centralized control, the update in SDN is able to be customized and carefully planned to avoid security vulnerabilities. We introduce a graph model to characterize the configuration update process, and propose a heuristic algorithm to find a safe update sequence with no security holes and has minimum cost. We evaluate the algorithm with experimental simulation, and the results show that it is effective when the traffic load is moderate and robust to different number of traffic classes and filter switches.

The remainder of this paper is organized as follows. In Section 2 we present the proposed SDN-based approach for security policies deployment. We show why configuration update would generate security holes through a simple example in Section 3. Then in Section 4, we introduce a safe update scheme by mathematically modeling the update process. After that, we evaluate the safe update scheme by experiment simulation in Section 5. We detail the related work in Section 6 and finally conclude the paper in Section 7.

2. Security policies deployment

2.1. Motivation

Generally, each security policy consists two parts, the target traffic and the corresponding action. The target traffic can be determined by a set of packet header fields, such as IP address, port and protocol. The action includes drop, pass, forward to a middlebox, etc. We show an example of security policies for one application in Table 1. In this example, users with IP addresses 123.4.1.* are untrusted users and therefore their SSH (dstPort = 22) and HTTPS (dstPort = 443) traffic should be dropped. On the other hand, users with IP addresses 115.6.2.* are authorized and all their traffic can pass through. Besides, the server with IP address 177.8.9.6 hosts important service or sensitive data and therefore packets forwarded to it have to pass the IDS at first. In

Table 2
The example of using flow entries to implement security policies.

srcIP	dstIP	dstPort	Action
123.4.1.*	*	22	Drop
123.4.1.*	*	443	Drop

addition to dropping or passing, there are other security policies like rate-limiting, e.g., restricting the bandwidth a VM can use for communicating with others, and communication isolation, e.g., restricting the VMs a specific VM can talk to [40].

We observe that the above abstraction of security policy is similar to the definition of flow entry in OpenFlow protocol [14]. In fact, each flow entry contains three parts, 1) the matching fields, which defines the patterns of headers in the packets that should be processed by the entry; 2) the action, which can be forward, drop, rewrite packets header, set queue or send to the controller; 3) statistics, which counts the number of packets and bytes matched this entry. Compared to the abstraction of security policy, the target traffic is similar to the matching filed, while the action of a security policy can be converted to the action of a flow entry. Therefore, the centralized controller can convert security policies into dedicated flow entries and install them on OpenFlow switches to enforce these policies.

It is efficient to use wildcard in matching field since the target traffic in a security policy always covers a set of possible packet headers. Table 2 shows the flow entries to implement the first policy of Table 1. The first flow entry matches packets, which have the destination port of 22 and source IP addresses of 123.4.1.*, and drops them. The second flow entry matches packets, which have destination port of 443 and source IP addresses of 123.4.1.*, and also drops them.

2.2. Design constraints and drawbacks of straightforward solutions

The converted flow entries need to be installed into switches to implement corresponding security policies. In large data centers, the number of these flow entries can be millions [39,40]. However, a single commodity switch can only provide a few thousand of flow entries [29,38]. Even though some high performance commercial products can support up to one million flow entries now, the traffic volume and the required number of flow entries may increase and finally excess the capacity of even the high performance switch. For example, Moshref et al. in [44] show that for data centers with 10K–100K VMs and 4K VLANS, it requires up to 400M rules to implement VLAN based traffic management. Therefore, it is not general to assume that all these flow entries can be deployed on a single high performance switch. Instead, they should be deployed on multiple switches.

There are several straightforward ways to install flow entries on multiple switches. One way is to divide the flow entries into several groups and install one group in one switch. Then to enforce the security policy, packets are steered through these switches one by one. It is apparent this way will prolong end-to-end delay and increase bandwidth consumption, and these overheads will increase as the total number of switches to store the flow entries grows. Therefore, it is not an efficient and scalable way.

Another way is to apply the security policies on edge switches such as Open vSwitch (OVS) or ToR (Top of Rack) switches. That is, put security policies related to the VMs in the same server, or the VMs connected to the same ToR switch together, and install them on OVS or ToR switch directly. However, Moshref et al. [44] have shown that OVS may consume a large fraction of CPU cycles, which are preferred to be allocated to client VMs by the operator to maximize their revenue [44]. On the other hand, if the operator constrains the CPU allocation to OVS, the number of flow entries and wildcard patterns that can be supported by OVS is limited and could be easily reached by some security policies [44]. Besides, the flow space of ToR switch is usually small and cannot always support all flow entries of VMs connected to it. Moreover, this way may lead to high control overhead since every time after the VM migration, the controller needs to update the flow entries in the OVS or ToR switches.

2.3. The proposed solution: an architecture with two layer OpenFlow switches

We have shown the drawbacks of two straight forward solutions. As a tradeoff, a better way is to put security policies with logical relationships together. For example, servers or VMs running the same application usually share the same security policy, such as the authorized source IPs, the accessibility to the database and etc., therefore the security policies of these servers or VMs can be put together and installed on the same switch. On one hand, it makes security policy management easier. On the other hand, it reduces the total number of flow entries required because servers running the same application can share one copy of flow entries. To obtain these benefits, we propose an architecture composed of two layer OpenFlow switches to implement the security policy, as shown in Fig. 1. The switches in the first layer classify the packets into different classes and forward them to one of the second layer switch. Each switch in the second layer implements security policies for one or more traffic classes. The classify switches add tags in packets header for the filter switches to distinguish packets of different traffic classes. We name the two layer switches classify switch and filter switch and use *S* and *F* to denote them respectively. In this architecture, each packet needs to go through only two switches, and hence the overhead of traffic redirection is mitigated. Besides, each filter switch needs to implement only part of the security policies, and consequently reduces the requirement of flow table size. Further, this architecture is more flexible since the installation of security policies is decoupled from the placement of VMs.

Now we introduce the details about traffic classification and load balance in the proposed architecture.

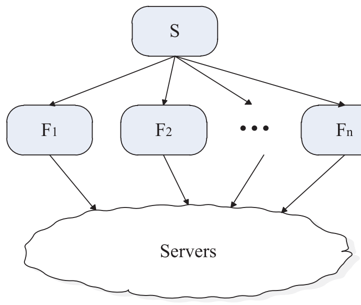

Fig. 1. Two layer architecture.

Traffic classification. A class of traffic denotes the packets with some common properties, e.g., the destination IP address belongs to the same subnet. The security policy of different traffic classes is different. For example, in data centers, each applications may have its own security policies. Thus, the classify switch can classify packets according to the application they belong to. Particularly, as the switch can identify different applications through the IP addresses of servers running them, the incoming packets can be classified based on destination IP address and outgoing packets can be classified based on source IP address. For multi-tenant data center, the tenant can specify its own security policies. Therefore the classification needs using one more filed to distinguish tenants, for example, VLAN ID.

Load balance. For some classes, the traffic volume may exceed the capacity of a single filter switch, and therefore need more than one filter switches to process the traffic. For other classes, the traffic volume is very small, and therefore some of them can share a single filter switch. The controller needs to take out load balance among different classes by assigning the limited filter switches to them. The authors in [34] provide a method to generate flow entries to forward the incoming packets to different servers according to load balance results, which can be used on switch S to carry out load balance in the proposed architecture.

Formally, considering n traffic classes denoted by $T_1, T_2, \ldots, T_n$. For packets in traffic class T_i, a set of security policies, denoted by R_i, should be applied on them. Switch S is configured to forward the incoming packets to one of the filter switches according to traffic classification and load balance results. Switch F_i is configured to carry out security policies of some traffic classes, according to the load balance results. If the traffic class of packets forwarded to a specific filter switch is included in the traffic classes whose security policies are carried out by that switch, the configuration is security satisfied, otherwise the configuration has security holes.

The proposed two-layer architecture significantly reduces the required number of flow entries in a single switch. Specifically, the classify switch only needs to contain flow entries to implement traffic classification, and each filter switch can contain flow entries to implement the security policies of only one traffic class. For example, considering deploying ACLs (Access Control Lists) for n applications, where the number of flow entries to implement the ACL for application i is l_i. Besides, supposing the traffic belong to different applications is isolated through different IP prefixes. Thus, switch S can classify the traffic into n classes according to IP prefix matching, which only needs n flow entries. Note that except IP prefix, other fields like VLAN can also be used for the isolation and the following results still apply. Further, each filter switch can deploy ACL of only one specific application. Therefore, the required number of flow entries in each filter switch is less than $\max\{l_1, l_2, \ldots, l_n\}$. Above all, the maximal number of required flow entries in a single switch is $\max\{l_1, l_2, \ldots, l_n, n\}$. In contrast, without the classify switch, all ACLs need to be installed in every edge switch at the worst case, and hence the required number of flow entries in a single switch is $l_1 + l_2 + \cdots + l_n$, which is much larger than $\max\{l_1, l_2, \ldots, l_n, n\}$ in most cases.

As a brief summary, the proposed approach is a logically centralized and physically distributed architecture for security policy enforcement, which employs multiple OpenFlow switches to implement security policies. In practical deployment, the classify switches should be placed at the ingress or egress point of the network, and the filter switches must be directly connected to them or indirectly connected to them through other switches. The operator has two options to deploy the proposed architecture, one is directly install new OpenFlow switches at the edge of the network according to the two layer architecture. The other is to embed the two-layer architecture in the existing OpenFlow switches. For example, making the core switches act as classify switches and the aggregation switches act as filter switches, and using packets encapsulation to redirect traffic from the classify switches to the filter switches.

2.4. Discussion

Although only one classify switch is illustrated in Fig. 1, it is not an essential requirement of the proposed architecture. Indeed, multiple classify switches can be deployed to avoid the possible single point bottleneck. When multiple classify switches are deployed, each classify switch is connected to all filter switches, and all classify switches use the same classification rules. Since these classify switches as whole implement classification and load balance functions, and they have the same classification rules, for simplicity consideration, we use a single switch to represent them in this paper.

Switch	Match Filed	Action
S	T_1 50%	Forward to F_1
	T_1 50%	Forward to F_2
	T_2 100%	Forward to F_3
F_1	SSH	Drop
F_2	SSH	Drop
F_3	SSH	Forward to L

(a)

Switch	Match Filed	Action
S	T_1 100%	Forward to F_1
	T_2 50%	Forward to F_2
	T_3 50%	Forward to F_3
F_1	SSH	Drop
F_2	SSH	Forward to L
F_3	SSH	Forward to L

(b)

Fig. 2. Example of flow table update (a) initial configuration, (b) final configuration.

In terms of scalability, adding more filter switches is able to improve the overall packets processing ability. However, only small scale data centers can be satisfied through this way since currently a commodity switch at most has hundreds ports. For larger data centers, it is practical to put multiple groups of the proposed two layer OpenFlow switches in parallel, so each group only needs to process a partial of the traffic. As different groups of OpenFlow switches can be treated independently, we only consider the scenario of single group in this paper.

3. Security holes when traffic changes

When traffic volume changes, the controller needs to recompute load balance and update flow tables in switches. Since different switches have different paths and delays to the controller, the controller cannot update flow tables in them at the same time [25], which may lead to security holes. Now we show this through a simple example.

Supposing there are two traffic classes T_1 and T_2. The security policy for T_1 is to drop SSH traffic, and the security policy for T_2 is to forward the SSH traffic to a SSL offload box, denoted as L. Initially, the traffic volume of these two classes are 60 and 30 respectively. Besides, there are three filter switches F_1, F_2, F_3 with the same capacity 100. For traffic class isolation and load balance concerns, S is configured to forward 50% of T_1's traffic to F_1, the other 50% to F_2, and all of T_2's traffic to F_3; F_1 and F_2 are configured to drop SSH traffic; F_3 is configured to forward SSH traffic to L. We show the configuration in Fig. 2(a).

Now consider the traffic volume of T_1 decreases to 50 and that of T_2 increases to 80. After recalculate the load balance, the controller decides to reconfigure the flow table to that shown in Fig. 2(b). However, unplanned update may lead to security holes. For instance, if the controller first modifies F_2's flow table to forward SSH traffic to L, it violates T_1's security policy as T_1's SSH traffic would be forwarded to L. On the other hand, if it first modifies S's flow table to forward packets according to the new configuration, it violates T_2's security policy as 50% of T_2's SSH traffic would be dropped.

One way to implement the above update without violating security policy is using the two-phase update method proposed by Reitblatt et al. [25]. Specifically, the controller first installs flow entries on F_2 to forward T_2's traffic to L, then updates flow table in S to forward all of T_1's traffic to F_1 and 50% of T_2's traffic to F_2, and finally removes flow entries of T_1 in F_2 at last. However, in this way, both T_1's and T_2's flow entries need to be stored on F_2 during the update. It implies that the filter switch F_2 must reserve at least the number of T_2's flow entries for the update, which is a waste of flow table resource as the time duration of the update is usually very short.

There is a more efficient way to implement the update. First, update S to forward all of T_1's traffic to F_1. Then, wait until F_2 completing processing in-flight packets and then update F_2 to forward SSH traffic to L. Finally, update S to forward 50% of T_2's traffic to F_2 and the other 50% to F_3. In this way, F_2 only needs to store either T_1's or T_2's flow entries, instead of both, which certainly reduces the resource overhead.

While the above update scheme is security guaranteed and resource efficient, it is not trivial to find such update schemes under general condition. Therefore, an efficient method to automatically find such schemes is required. We propose a graph model based method and introduce it in next section.

4. Security policies guaranteed update scheme

4.1. Problem formulation

Given the number of traffic classes, traffic volume of each class, and the number of filter switches, there is a variety of configurations to allocate filter switches to these traffic classes and distribute traffic among allocated filter switches. Each configuration can be regarded as a node, while configuration update can be regarded as the transition from one node to another node, i.e., an edge connecting these two nodes. Therefore, a graph can be used to model the possible configurations and updates. Based on the graph, an update scheme from the initial configuration to the final configuration equals to a path from the initial node to the final node. Security policies guaranteed update means that the corresponding configuration of each node on the path satisfies all security policies. Next we will detail the definition of our graph model.

Node. Consider m traffic classes, denoted as $T_1, T_2, \dots, T_m$, and n filter switches, denoted as $F_1, F_2, \dots, F_n$. The security policy of T_i $(i = 1, 2 \dots, m)$ is denoted as R_i, and the capacity of F_j $(j = 1, 2, \dots, n)$ is denoted as C_j. Further, we use T to denote traffic classes set, F to denote filter switches set, R to denote security policies set and C to denote capacity set. Then, we define W as a volume function $W : T \to R^+$, where $W(T_i)$ gives the total volume of T_i. We also define a forward function $P : F \to T \times R^+$ where

$P(F_i) = (T_j, c)$ means that S is configured to forward c unit of T_j's traffic to F_i. We then define a policy mapping function M: $F \times T \to \{0, 1\}$, where $M(F_i, T_j) = 1$ implies that the security policy of traffic class T_j is installed in F_i. With these definitions, we use the seven-tuple (T, F, R, C, W, P, M) to represent a configuration state, and regard it as a node in our graph model.

Edge. Generally, a directed edge from one node to another node represents the update of flow tables to transit the configuration from one state to another. Our proposed architecture contains two general types of update in a single step. One is modifying S's flow table to change packet forwarding rules, which induces the change of forward function P; the other is updating the flow table of a filter switch to change the security policies it carries, which induces the change of mapping function M. Therefore, we define the edge set as follows:

$$l(v_a, v_b) = \begin{cases} 1 & \text{if } M_a = M_b \text{ or } P_a = P_b, \quad \sum_{i=1}^{n}\sum_{j=1}^{m} I_{M_a(F_i,T_j)\neq M_b(F_i,T_j)} = 1 \\ 0 & \text{else.} \end{cases} \tag{1}$$

where we assume $v_a = (T, F, R, C, W, P_a, M_a)$, $v_b = (T, F, R, C, W, P_b, M_b)$. $l(v_a, v_b) = 1$ means that there is a directed edge from v_a to v_b. The conditions of $M_a = M_b$ represent that v_a and v_b have the same configurations in all filter switches, but different configuration in classify switch; and the conditions of $(P_a = P_b, \sum_{i=1}^{n}\sum_{j=1}^{m} I_{M_a(F_i,T_j)\neq M_b(F_i,T_j)} = 1)$ represent that v_a and v_b have the same configuration in all switches except one filter switch. I is the indicate function.

Security policies guaranteed update. Note that only some configuration are security satisfied. In these configuration, if the traffic of some classes is forwarded to a filter switch, then the filter switch must implement the security policy of these classes. Translating it into a mathematical proposition, that is: $\forall\, i = 1, 2, .., n,$ if $P(F_i) = (T_j, c)$ and $c \neq 0$, then $M(F_i, T_j) = 1$. We use V to denote configuration states satisfying this proposition, and refer it as security satisfied configuration set. For security guaranteed update, every middle configuration states must belong to V. In other words, if the update process only involves the configuration states that belong to V, the security will be guaranteed. Based on this observation, we directly use V as the node set in our graph model.

Edge weight definition. With all of the above definition, we get a directed graph $G = (V, E(V))$, where V is the set of security satisfied configuration states, and $E(V)$ is the edge set defined on $V \times V$ according to (1). Assuming that the initial configuration v_i and the final configuration v_f are both security satisfied, then finding a security policies guaranteed update scheme equals to finding a path from v_i to v_f in G. We assign each edge a weight to characterize the update cost, and then select the path with minimum total cost as the optimal path.

We consider three types of cost caused by a single update from v_a to v_b ($v_a = (T, F, R, C, W, P_a, M_a)$, $v_b = (T, F, R, C, W, P_b, M_b)$). The first type is about flow entry update, reflecting the cost caused by the controller operating flow tables. We use a constant C_m to describe it for simplicity. The second type is about packet loss. When the volume of traffic forwarded to a filter switch exceeds its capacity, the excess packets would be dropped and thus the network service would be degraded. We use C_l to denote the cost of this type, which is defined as the average of the excess traffic of v_a and v_b, as shown in (2). The third type is about load balance. The average of traffic volume is used to carry out the load balance, while in reality, the traffic volume changes all the time. A bad load balance will degrade the system's ability to deal with burst traffic. We exploit the variance of filter switches utilization to describe the load balance level, and define the unbalanced cost C_u as the average of load balance level of v_a and v_b, as shown in (3). $(P_i(F_j)^{(2)}$ represents the second dimension of $P_i(F_j)$, $i = a, b$).

$$C_l(v_a, v_b) = \frac{1}{2}\sum_{j=1}^{n}\left([P_a(F_j)^{(2)} - C_j]^+ + [P_b(F_j)^{(2)} - C_j]^+\right), \tag{2}$$

$$C_u(v_a, v_b) = \frac{1}{2}(\text{Variance}(a) + \text{Variance}(b)), \tag{3}$$

$$\text{Variance}(i) = \frac{1}{n}\sum_{j=1}^{n}\left(\frac{P_i(F_j)^{(2)}}{C_j} - \frac{1}{n}\sum_{j=1}^{n}\frac{P_i(F_j)^{(2)}}{C_j}\right)^2. \tag{4}$$

Given three types of cost, the weight can be defined in a variety of ways, we simply use the weighted sum of three individual cost as the final weight of edge from v_a to v_b:

$$w(v_a, v_b) = C_m + \alpha C_l(v_a, v_b) + \beta C_u(v_a, v_b). \tag{5}$$

4.2. Problem solution

Theoretically, the Dijkstra algorithm can be used to find the shortest path from the initial state v_i to the destination state v_f. However, the number of states could be infinite with a given mapping function M. So practically, it needs to restrict forward function to make the state number finite. Further, the total number of states could still be large even after the restriction, since the number of mapping function M is huge (2^{mn} totally). Therefore, it is hard to exploit Dijkstra algorithm to find the path. Instead, sometime efficient algorithms are needed to approximate the optimal result. Now, we introduce our strategy to reduce the number of states, and a heuristic algorithm to get a sub-optimal result.

4.2.1. *State set reduction*

Note that every configuration state in V has the property that $\forall\ i = 1, 2, .., n$, if $P(F_i) = (T_j, c)$ and $c \neq 0$, then $M(F_i, T_j) = 1$, which means the value of $M(F_i, T_j)$ is totally decided by $P(F_i)$ when $c \neq 0$. That is to say, if two states have the same mapping function, their difference lies only on allocation of traffic to filter switches. However, the possible allocation is infinite if no constraints are deployed. But in consideration of configuration update, we change forwarding functions for two purposes, one is steering traffic forwarded to one filter switch to other filter switches, then the controller can modify the security policy on that switch; the other is reallocating traffic to the filter switches to achieve better load balance. Further, the operation for above two purposes is always successive, i.e., the traffic belong to the same class will be proportionally allocated to filter switches implementing the corresponding security policies. Based on these analysis, the forwarding function should satisfy the following restriction.

$$I_{ij} = \begin{cases} 1 & \text{if } P(F_i) = (T_j, c) \text{ and } c \neq 0 \\ 0 & \text{else,} \end{cases} \tag{6}$$

$$D_j = \sum_{j=1}^{n} I_{ij} C_i, \tag{7}$$

$$P(F_i) = \begin{cases} (T_j, \frac{M(T_j)}{D_j} C_i) & \text{if } I_{ij} = 1 \text{ and } D_j \neq 0 \\ (*, 0) & \text{else.} \end{cases} \tag{8}$$

The D_j in equality (7) represents the aggregate capacity of filter switches who are configured to carry out the security policy of T_j, and (8) shows the proportional allocation of T_j's traffic to these switches.

4.2.2. *Heuristic algorithm*

We show the proposed heuristic algorithm in Algorithm 1. We use v_i and v_f to represent the initial and final state. As at the

Algorithm 1 Heuristic algorithm for security guaranteed update.

```
1: initialize $v_i = (W_f, P_i, M_i), v_f = (W_f, P_f, M_f)$
2: $F_{change} \Leftarrow NULL$
3: for $i = 1$ to $n$ do
4:   for $j = 1$ to $m$ do
5:     if $M_i(F_i, T_j) \neq M_f(F_i, T_j)$ then
6:       Add $F_i$ to $F_{change}$
7:     end if
8:   end for
9: end for
10: while $F_{change} \neq \emptyset$ do
11:   $F^* \Leftarrow F$ in $F_{change}$ with minimum $Cost(F)$
12:   $Update(F^*, v_i, v_f)$
13:   Remove $F^*$ from $F_{change}$
14:   Output $F^*$
15: end while
```

beginning of the update, the traffic volume has changed, so v_i and v_f have the same traffic volume function W. Besides, we assume traffic class T, filter switches F and their capacity C does not change and exclude them from the state representation. Then in steps 3–7, we select filter switches whose security configuration changes and denote them as F_{change}. Steps 8–13 circularly update switches in F_{change}. In each cycle period, the algorithm selects one switch from F_{change} according to the update cost. Specifically, the cost to update switch F, i.e., change F's configuration from $M_i(F)$ to $M_f(F)$, is calculated based on the update process, which is represented by Update(F, v_i, v_f) and contains three concrete steps:

1. Clear traffic forwarded to F, that is modifying S's configuration to forward the traffic forwarded to F to other switches. When there are multiple classify switches, the controller needs to modify all these switches' configuration. From Section 4.2.1 we know that the traffic allocation after this step is unique. Besides, the controller computes the cost in this step according to (5) and we use $C_{\text{clear}}(F)$ to denote it.
2. Change F's configuration to carry out the security policy according to M_f, i.e., installing security policies of T if $M_f(F, T) = 1$. As the traffic forwarded to F is zero after the last step, it is safe to change F's configuration. We use $C_{\text{change}}(F)$ to denote cost in this step.
3. Modifying S's configuration to forward packets to F according to M_f, i.e., forward packets in traffic class T to F if $M_f(F, T) = 1$. It is safe to forward packets according to M_f now as F has been configured according to M_f. We use $C_{\text{forward}}(F)$ to denote cost in this step.

Table 3
Default parameters of experiment simulation.

Symbol	Definition	Default value
n	Number of filter switches	24
m	Number of traffic classes	5
C	Capacity of filter switch	100
ρ	Average traffic load	0.6
$\Delta\rho$	Average traffic load variance	0.2
α	The weight of packet loss cost	1000
β	The weight of load balance cost	1
C_m	The modify cost	0

Then, the cost to update F, denoted as $\text{Cost}(F)$, is characterized as the sum of cost in above three steps:

$$\text{Cost}(F) = C_{\text{clear}}(F) + C_{\text{change}}(F) + C_{\text{forward}}(F). \tag{9}$$

With this formula, the algorithm calculates the cost to update each switch and selects the one with minimum cost, denoted as F^*, and then updates F^* according to the above steps. After that, it removes F^* from F_{change} and enters to next cycle.

Note that when there are multiple classify switches, updating these switches' configuration to change the traffic forwarded to F in the first and third step may cause transient inconsistency, i.e., some of them have finished the update while others have not. However, this will not cause security holes. Take the first step as example, since F's configuration is not changed during this step, if S has not finished the update, the traffic it forwards to F is the same as the original, and thus is security guaranteed; if S has finished the update, the traffic it forwards to F will be zero, and thus is also security guaranteed. Similar techniques can be applied to explain that updating multiple classify switches in the third step will not cause security holes.

5. Simulation experiments

5.1. Simulation setup

Traffic volume distribution. In our simulation, we assume all filter switches have the same capacity and each filter switch can only be assigned to one traffic class. We also assume that the traffic volume of each class follows the same uniform distribution. Specifically, we use ρ to denote the average traffic load, $\Delta\rho$ to denote the average traffic load variance, C to denote the capacity of a filter switch, m to denote the number of traffic classes and n to denote the number of filter switches. Then the traffic volume of each class is generated with uniform distribution in interval of $[(\rho - \Delta\rho)\frac{nC}{m}, (\rho + \Delta\rho)\frac{nC}{m}]$.

Load balance algorithm. We use a simple but effective load balance scheme to assign n filter switches to m traffic classes in our simulation. At first, for each traffic class, we assign a least number of filter switches that is enough to support its traffic volume. Then, we assign the rest switches in an iterative way. In each iteration, we first calculate current traffic load of each class, which is defined as the ratio of traffic volume of that class over the total capacity of filter switches assigned to it. Then, we assign a rest switch to the traffic class who has the highest traffic load.

Edge weight related settings. We set $\alpha = 1000$ and $\beta = 1$ to stress the cost of packet loss during the update. Besides, as we assume every update step has the same modify cost, C_m has no influence to path selection in our heuristic algorithm, therefore we set it to 0.

Performance metrics. We use load balance level and percentage of packet loss as the performance metrics. The load balance level is defined as the square root of average load balance cost in each update step, which reflects the difference of filter switches' utilization. The percentage of packet loss is defined as the ratio of average packet loss cost in each update step over the total traffic volume. We measure them separately and average them with 1000 round simulations to reduce the influence of randomness.

We implement a simulator in MATLAB according to above settings. The simulator first generates the initial traffic volume of each class, then assigns the filter switches to these classes using the above load balance algorithm. After that, the simulator obtains the initial configuration of the classify and filter switches, and repeats the above procedure to obtain the final configuration of the switches. Finally, it executes the update using the proposed algorithm, with the initial and final configuration as inputs, and measures the performance results. We do simulations with different number of filter switches (n), and observe how traffic load and the number of traffic classes would influence the performance. Table 3 lists the default value of parameters in our simulation, we will use them if not noted otherwise.

5.2. The influence of traffic load

We first study how traffic load influences the performance. As the traffic load variance ($\Delta\rho$) is set to 0.2, to make the traffic volume distribution of $[(\rho - \Delta\rho)\frac{nC}{m}, (\rho + \Delta\rho)\frac{nC}{m}]$ valid and meaningful, the value of average traffic load ρ is restricted to [0.2, 0.8]. Fig. 3 shows the performance with different traffic load ρ. As the figure illustrates, generally, when the traffic load increases, both the load balance level and the percentage of packet loss have an increasing tendency. An exception is in some cases (when $n = 48, 64, 128$) the load balance level is lower when the traffic load is 0.8 compared to that when the traffic load is 0.7. We

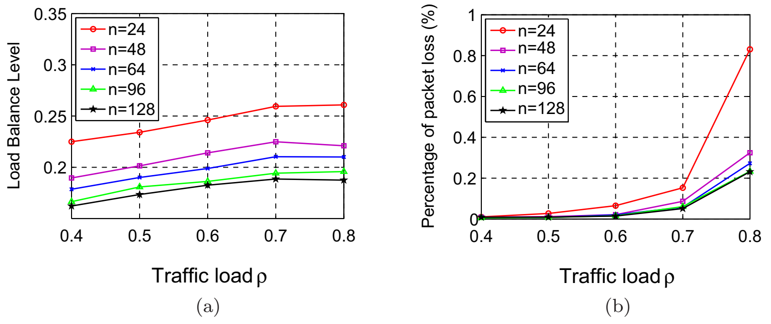

Fig. 3. Different metrics of performance during the update under different traffic loads. (a) Load balance, (b) packet loss.

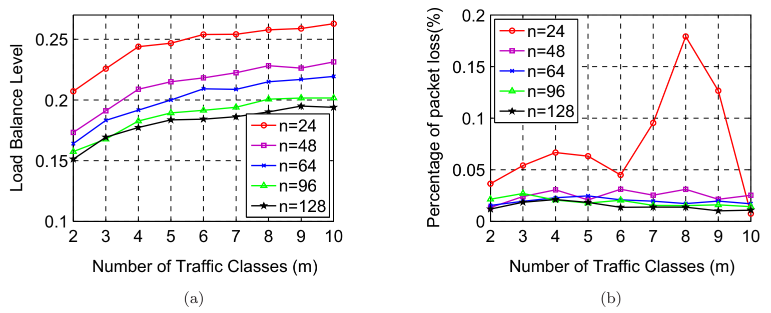

Fig. 4. Different metrics of cost during the update with different number of traffic classes. (a) Load balance, (b) packet loss.

checked the simulation data and found this was because when the traffic load was 0.8, the filter switches were more likely to be saturated during the update. As the utilization of a saturated switch is 1, so when more switches become saturated, the variance of utilization becomes smaller. The figure also shows increasing the number of filter switches would not degrade the performance. On the contrary, the performance becomes better in most cases. Besides, the percentage of packet loss is low on general, especially when the traffic load is low, for example, less than 0.2% packets will be dropped when the traffic load is less than 0.7 (note that the *y*-axis in Fig. 3 is expressed as percentage in linear function, e.g., the value 0.2 represents 0.2% packet loss). As far as we know, the traffic load in real networks is low on average, so the proposed update scheme can be applied on them with only little packet loss.

5.3. The influence of number of traffic classes

We then study how the number of traffic classes influence the performance. In Fig. 4, we present the performance results with different number of traffic classes. From Fig. 4, we can observe that when the number of traffic classes increases, the performance of load balance becomes worse, while the performance of packet loss fluctuates. We explain the reasons as follows. For load balance, when the number of traffic classes increases, it will become worse as each filter switch can only be assigned to one traffic classes. Therefore, the performance of load balance during the update also become worse. For packet loss, we can observe an obvious spike when the number of filter switches is 24 and the number of traffic classes is 8. After analyzing the output of the simulator, we find that the spike is caused by the influence of two factors. The first is the ratio of update steps that have packet loss, which is 0.031, 0.0205, 0.085, and 0.124 when the number of traffic classes is 6, 7, 8, and 9 respectively. We can find that this ratio is larger when the number of traffic classes is 8 compared to that of 6 and 7 traffic classes, which suggests that packet loss occurs more frequently when the number of traffic classes is 8. While this ratio is even larger when the number of traffic classes is 9, the final average packet loss is lower due to the influence of the other factor—the average packet loss in an update step that includes packet loss. The value of this metric is 8.75%, 28.52%, 17.74%, and 6.74% when the number of traffic classes is 6, 7, 8, and 9 respectively. These values suggest that if packet loss occurs in an update step, it is more sever when the number of traffic classes is 8, and thus the overall packet loss is higher.

By comparing Fig. 3 and Fig. 4, we can also find that the performance change caused by the number of traffic classes is rather low, which suggests that the update algorithm is robust to different number of traffic classes. Further, the performance becomes better in most cases when the number of filter switch increases, which is in accordance with the result in Section 5.2. Because when there are more filter switches, it is more possible to steer traffic forwarded to one switch to another without any packet loss, and clearing traffic forwarded to a single switch (that means the utilization of the switch will become zero) will lead to smaller degradation on the load balance.

For data centers, the number of traffic classes may be more than 10. Therefore, we also run simulations with more traffic classes and filter switches. Table 4 shows the results with 100, 200, 300 and 400 traffic classes, in which the number of filter

Table 4
Simulation results (number of filter switches: 1000, average traffic load: 0.7).

Number of traffic classes	100	200	300	400
Percentage of traffic loss (%)	0.018	0.021	0.035	0.323
Load balance level	0.210	0.210	0.220	0.235

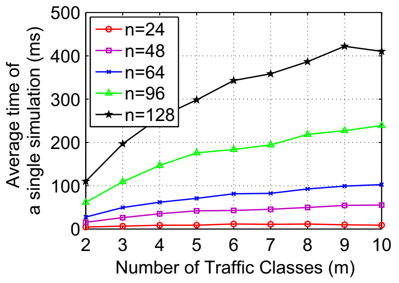

Fig. 5. Time cost with different number of traffic class.

switches is set to 1000 and the average traffic load is set to 0.7. From these results, we can find that the performance results is similar to that with fewer traffic classes. For example, when there are 400 traffic classes, the percentage of traffic loss is 0.323% and the load balance level is 0.235. These results further suggest that the number of traffic classes has little influence on the performance of the update.

To quantitatively analyze the complexity of our algorithm, we measure the time cost of 1000 round simulation with different number of traffic classes and plot the result in Fig. 5. A PC with 3.2 GHz Intel i5 CPU is used for simulation. As the number of filter switches increases, the complexity of the update problem increases, therefore the time cost increases, just as the figure illustrates. Even though, we can see that the time spent for a single simulation is little in general, less than 500 ms in our simulation. These results suggest that the proposed update method can be deployed on the controller to execute in real time.

6. Related work

Security policies enforcement. Middleboxes such as firewalls and DPIs are widely used to implement security policies. Joseph et al. in [9] proposed to connect middleboxes to policy aware switches and configure these policy aware switches to forward packets to different middleboxes. Qazi et al. [41] further proposed to use SDN controller to install flow entries in switches to automatically steer packets to different middleboxes for policy enforcement. In contrast to these works, we explore another approach that exploits OpenFlow switches for security policies enforcement by installing specific flow entries. Particularly, in consideration of limited size of flow table in a single OpenFlow switch, we propose a scalable and efficient architecture with two-layer OpenFlow switches to implement the security policies.

Many previous works have also addressed the problem of updating middlebox configuration [9,18–21]. Zhang et al. [18] illustrated that updating firewall configuration would generate security holes, i.e., allowing some illegal traffic passing through or blocking some legal traffic. They proposed a two-phase update scheme to guarantee safety during the update. Kartit et al. [19] further proposed an accurate algorithm for Type I update introduced in [18]. In [20,21], methods were proposed to analyze firewall configurations to find out the possible anomalies and correct them. We also studied configuration update in this paper, but with different problem and solution. Specifically, our problem is to update the configuration of OpenFlow switches that implement security policies for better load balance, rather than modifying rules in firewalls. Besides, we propose a heuristic algorithm to carry out the update, with the guarantee of no security hole caused by the update.

Consistent network update. Many works have studied transient behaviors during update of traditional networks [3,4,28,31]. Most of them aimed at avoiding routing loops [4,28] and loss of connectivity [3] during data plane update. Vanbever et al. [31] also studied anomalies during control plane update, such as IGP protocol replacement and reconfiguration. While the update problem studied in this paper can also be categorized into data plane update, it has a different background (SDN) and objective (to avoid security holes) with these works.

More recent works focus on network update in SDN. Reitblatt et al. [26] proposed a two-phase update scheme to implement the consistent update. In the proposed scheme, the switches are installed with flow entries of both initial and final configuration during the update, and they exploit the tag, added at the ingress switch, to determine which set of flow entries to be used to process an incoming packet. Reitblatt et al. [25] proved that the two-phase update scheme can guarantee per-packet and per-flow consistency. The two-phase update scheme is general and can provide security guaranteed update under scenarios considered in this paper. However, to be compatible with current network protocol, it requires a public field of packets header to act as the tag, which increases the complexity of network management and restricts its deployment in real network environment. Besides, the switches need to store flow entries of both the initial and final configuration. To achieve this, the switches need to reserve some flow entries for the update in two-phase update scheme, which is a waste of flow table resource since the update procedure only lasts a very short time. In this paper, we avoid the reservation by leveraging the same set of flow entries in other switches.

In [45], Canini et al. explored the problem of consistent composition of concurrent policies update. They proposed that the controller should provide interfaces for the application to execute policy update in a transactional way, i.e., the update is either committed or aborted. While the proposed transactional interface can be used to update the configuration of filter switches, its implementation is based on two-phase update method [25] and thus is also inefficient in terms of flow table resource.

To achieve the similar consistency in [26], McGeer [13] proposed to forward the packets affected by the update to the controller. This scheme does not require the tags and reservation of flow entries. However, it requires more controller resource, for example, the buffer size and inbound bandwidth. Under the scenario of packet filtering, the volume of traffic affected by the update may be large. Therefore this scheme is not a scalable solution and we do not take it in this paper.

Sequence planning for network update. Some works have studied the problem of sequence planning for network update under different scenarios. Raza et al. [24] studied how to plan the sequence to carry out links weight update, as well as the sequence to deactivate and reactivate links for maintenance in multi-path routing. Józsa and Makai in [10] investigated the reroute sequence planning problem in MPLS networks. Ghorbani and Caesar in [5] explored the sequence planning problem for VM migration. The proposed heuristic algorithm in this paper is inspired by models and solutions of these works, but we mainly focus on the sequence to safely update OpenFlow switches, which is orthogonal to these previous works.

7. Conclusion

Since traditional network uses distributed control, security enforcement and configuration update are complicated. Through separating control plane from data plane and employing centralized control, software-defined networking provides an opportunity to address these issues. In this paper, by leveraging the advantage of SDN, we study another scheme that directly installs flow entries in OpenFlow switches for security policy enforcement. Specifically, we propose an efficient and scalable architecture with two layer OpenFlow switches for deploying security policies. Moreover, we illustrate that security holes would be generated during updating the configuration of these switches. To avoid it, we propose a security guaranteed update scheme based on a graph model. Compared with existing update schemes, the proposed scheme requires smaller size of flow table and has lower overhead on the control plane. Furthermore, we carry out a thorough evaluation for the proposed scheme, which demonstrates that it is effective and robust under an extensive range of conditions.

Acknowledgments

This work is supported by National Basic Research Program of China (973 Program) (No. 2013CB329105), National Nature Science Foundation of China (grant nos. 61301080, 91338203, 91338102, and 61321061).

References

[1] M. Arregoces, M. Portolani, Data center fundamentals, Cisco Press, 2003.
[2] Enterprise Network and Data Security Spending shows Remarkable Resilience. http://en.wikipedia.org/wiki/Softwaredefined_data_centerCurrent_status.
[3] P. Francois, O. Bonaventure, B. Decraene, P.-A. Coste, Avoiding disruptions during maintenance operations on BGP sessions, IEEE Trans. Netw. Serv. Manage. 4 (3) (2007) 1–11.
[4] J. Fu, P. Sjodin, G. Karlsson, Loop-free updates of forwarding tables, IEEE Trans. Netw. Serv. Manage. 5 (1) (2008) 22–35.
[5] S. Ghorbani, M. Caesar, Walk the line: consistent network updates with bandwidth guarantees, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, ACM, 2012, pp. 67–72.
[6] Y. Gong, Z. Ying, M. Lin, A survey of cloud computing, in: Proceedings of the Second International Conference on Green Communications and Networks, Springer, 2013, pp. 79–84.
[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: towards an operating system for networks, ACM SIGCOMM Comput. Commun. Rev. 38 (3) (2008) 105–110.
[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, A. Vahdat, B4: experience with a globally-deployed software defined WAN, in: Proceedings of SIGCOMM 2013 Conference on SIGCOMM, ACM, 2013.
[9] D.A. Joseph, A. Tavakoli, I. Stoica, A policy-aware switching layer for data centers, ACM SIGCOMM Comput. Commun. Rev. 38 (2008) 51–62.
[10] B.G. Józsa, M. Makai, On the solution of reroute sequence planning problem in MPLS networks, Comput. Netw. 42 (2) (2003) 199–210.
[11] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, S. Shenker, ONIX: a distributed control platform for large-scale production networks, in: Proceedings of the Ninth USENIX Conference on Operating Systems Design and Implementation, 2010, pp. 1–6.
[12] J.F. Kovar, The 10 Biggest Data Center Stories of 2012. http://www.crn.com/slide-shows/data-center/240143993/the-10-biggest-data-center-stories-of-2012.htm?pgno=7, 2012.
[13] R. McGeer, A safe, efficient update protocol for openflow networks, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, ACM, 2012, pp. 61–66.
[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[15] Openflow Specification Version 1.1.0. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.
[16] Openflow@google. http://www.opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf.
[17] M. Prigge, Understanding the Software-defined Data Center, InfoWorld.
[18] C.C. Zhang, M. Winslett, C.A. Gunter, On the safety and efficiency of firewall policy deployment, in: IEEE Symposium on Security and Privacy, 2007, pp. 33–50.
[19] A. Kartit, A. Radi, M.E. Marraki, B. Regragui, Safe and correctness strategies for updating firewall policies, Int. J. Comput. Sci. Netw. Secur. 11 (3) (2011) 15.
[20] E. Al-Shaer, H. Hamed, R. Boutaba, M. Hasan, Conflict classification and analysis of distributed firewall policies, IEEE J. Selected Areas Commun. 23 (10) (2005) 2069–2084.
[21] S. Halle, E.L. Ngoupé, R. Villemaire, O. Cherkaoui, Distributed firewall anomaly detection through LTL model checking, in: IFIP/IEEE International Symposium on Integrated Network, 2013, pp. 194–201.
[22] C. Yoon, T. Park, S. Lee, H. Kang, S. Shin, Z. Zhang, Enabling security functions with SDN: a feasibility study, Comput. Netw. 85 (0) (2015) 19–35.

[23] A. Ding, J. Crowcroft, S. Tarkoma, H. Flinck, Software defined networking for security enhancement in wireless mobile networks, Comput. Netw. 66 (0) (2014) 94–101.
[24] S. Raza, Y. Zhu, C.-N. Chuah, Graceful network state migrations, IEEE/ACM Trans. Netw. 19 (4) (2011) 1097–1110.
[25] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, in: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 323–334.
[26] M. Reitblatt, N. Foster, J. Rexford, D. Walker, Consistent updates for software-defined networks: change you can believe in!, in: Proceedings of the 10th ACM Workshop on Hot Topics in Networks, ACM, 2011.
[27] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, OpenFlow Switch Consortium (2009). Technical report.
[28] L. Shi, J. Fu, X. Fu, Loop-free forwarding table updates with minimal link overflow, in: Proceedings of IEEE International Conference on Communications, 2009.
[29] B. Stephens, A. Cox, W. Felter, C. Dixon, J. Carter, Past: Scalable ethernet for data centers, in: Proceedings of the Eighth International Conference on Emerging Networking Experiments and Technologies, ACM, 2012, pp. 49–60.
[30] The Software Defined Data Center-networking. http://www.vmware.com/software-defined-datacenter/networking-security.html.
[31] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, O. Bonaventure, Seamless network-wide IGP migrations, in: Proceedings of the ACM SIGCOMM 2011 Conference, 2011, pp. 314–325.
[32] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, S. Shenker, Middleboxes no longer considered harmful, in: Proceedings of the Fifth USENIX Conference on Operating Systems Design and Implementation, 2004, p. 15.
[33] K. Wang, Y. Qi, B. Yang, Y. Xue, J. Li, LiveSec: towards effective security management in large-scale production networks, in: 2012 IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), 2012, pp. 451–460.
[34] R. Wang, D. Butnariu, J. Rexford, Openflow-based server load balancing gone wild, in: Proceedings of the 11th Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, USENIX Association, 2011, p. 12.
[35] M. Ali, S.U. Khan, A.V. Vasilakos, Security in cloud computing: opportunities and challenges, Inf. Sci. 305 (2015) 357–383.
[36] V. Varadharajan, U. Tupakula, Security as a service model for cloud environment, IEEE Trans. Netw. Serv. Manage. 11 (1) (2014) 60–75.
[37] L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, A. Vasilakos, Security and privacy for storage and computation in cloud computing, Inf. Sci. 258 (2014) 371–386.
[38] D. Kreutz, F. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: a comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76.
[39] M. Moshref, M. Yu, A. Sharma, R. Govindan, Scalable rule management for data centers, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2013, pp. 157–170.
[40] L. Popa, M. Yu, S.Y. Ko, CloudPolice: taking access control out of the network, in: Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.
[41] Z.A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, M. Yu, SIMPLE-fying middlebox policy enforcement using SDN, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, 2013, pp. 27–38.
[42] S. Subashini, V. Kavitha, A survey on security issues in service delivery models of cloud computing, J. Netw. Comput. Appl. 34 (2011) 1–11.
[43] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, V. Sekar, Making middleboxes someone else's problem: network processing as a cloud service, ACM SIGCOMM Comput. Commun. Rev. 42 (4) (2012) 13–24.
[44] M. Moshref, M. Yu, A. Sharma, R. Govindan, vCRIB: virtualized rule management in the cloud, in: Proceedings of NSDI, 2013.
[45] M. Canini, P. Kuznetsov, D. Levin, S. Schmid, Software transactional networking: concurrent and consistent policy composition, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, ACM, 2013, pp. 1–6.